Aspects of Data Warehouse Technologies for Complex Web Data

Thomsen, Christian

Publication date:
2008

Document Version
Publisher's PDF, also known as Version of record

Aspects of Data Warehouse Technologies for Complex Web Data

Christian Thomsen

Ph.D. Dissertation

A dissertation submitted to the Faculties of Engineering, Science and Medicine at Aalborg University, Denmark, in partial fulfillment of the requirements for the Ph.D. degree in computer science.

Abstract

This thesis is about aspects of specification and development of data warehouse technologies for complex web data. Today, large amounts of data exist in different web resources and in different formats. But it is often hard to analyze and query the often big and complex data or data about the data (i.e., metadata). It is therefore interesting to apply Data Warehouse (DW) technology to the data. But to apply DW technology to complex web data is not straightforward and the DW community faces new and exciting challenges. This thesis considers some of these challenges.

The work leading to this thesis has primarily been done in relation to the project European Internet Accessibility Observatory (EIAO) where a data warehouse for accessibility data (roughly data about how usable web resources are for disabled users) has been specified and developed. But the results of the thesis can also be applied to other projects using business intelligence (BI) and/or complex web data. An interesting perspective is that all the technologies used and developed in the presented work are based on open source software.

The thesis presents several tools in a survey of the possibilities for using open source software for BI purposes. Each category of products is evaluated against criteria relevant to the use of BI in industry. After this, experiences from designing and implementing a DW for accessibility data are presented. Further, the conceptual, logical, and physical models for the DW are presented. This is believed to be the first time a general and scalable DW is built for the accessibility field which is both complex to model and to calculate aggregation results for.

The thesis then presents solutions to general interesting problems found during the work on developing a DW and supporting DW technologies for the EIAO project. A new and efficient way to store triples from an OWL ontology known from the Semantic Web field is presented. In contrast to traditional triple stores where the data is stored in few, but big, tables with few columns, the presented solution spreads the data over more tables that may have many columns. This makes it efficient to insert and extract data, in particular when using bulk loading where big amounts of data are considered.

A new and flexible way to exchange relational data via the XML format (which is, e.g., used by web services) is also presented. This method saves labor to program often complex solutions to handle correct exchange of data. With the presented method, the user only has to specify what data to export and the structure of the generated XML. The data can then automatically be exported to XML and imported into another database just like updates to the XML automatically can be migrated back to the original database.

Regression test is widely accepted and used in traditional software development. For Extract–Transform–Load (ETL) software, regression test is, however, traditionally cumbersome and time-consuming. The thesis points out crucial differences between test of "normal" software and ETL software and on that background a new semi-automatic framework for regression test of ETL software is introduced. The framework makes it easy and fast to start doing regression test. It only takes minutes to set up regression test with the framework.

Traditionally DWs have been bulk loaded with new data at regular time intervals, e.g., monthly, weekly, or daily. But a new trend is to add new data as soon as it becomes available from, e.g., a web log or another online resource. This is done by means of SQL INSERT statements but these are slow compared to bulk loading techniques and the performance of the database systems drops. Therefore the thesis presents a new and innovative method that combines the best of these worlds. Data can be made available in the DW exactly when needed and the user gets bulk-load speeds, but INSERT-like data availability.

Acknowledgments

There are many people I would like to thank for their help and support during my Ph.D. project. First of all, my thanks go to my Ph.D. supervisor, Torben Bach Pedersen, for his great interest in and support of my work during the entire Ph.D. project. By his own good example, he has taught me how to do research and turn loose and/or wild ideas into concrete work.

During my Ph.D project, I visited Dresden University of Technology for four months. I would like to thank Professor Wolfgang Lehner and all members of his group for hosting and integrating me in their group and for the collaboration and support. Further, I thank them for all the nice excursions to beautiful places in Saxony after work.

Thanks also go to my colleagues in the Database and Programming Technologies Group at Aalborg University for a good and motivating working environment. I also thank the many people I have worked with in the European Internet Accessibility Observatory project and in particular the software developers with whom I have had many fruitful and interesting technical discussions.

Last, but definitely not least, I would like to thank my girlfriend Maria and my family for all their support and encouragement. To have such a good home base made my work much easier.

This work was supported by the European Internet Accessibility Observatory (EIAO) project, funded by the European Commission under Contract no. 004526.

Contents

Chapter 1

Introduction

Today, the Web is the biggest available information source. The Web is used daily by hundreds of millions of people and in many countries nearly all companies, authorities, and organizations have web sites. Many things that used to be time consuming and cumbersome such as finding and booking cheap flights, finding pictures of a specific Aboriginal painter, or obtaining statistics about the the economic growth in each of the EU countries can today be done in minutes or even less from a computer connected to the Web. The popularity of the Web thus gives new possibilities, but it also introduces new challenges.

Huge amounts of informations are available in a variety of different formats on the Web. Much information is represented in the HyperText Markup Language (HTML) format used for web pages, but the broad use of the Web has also led to other formats such as Extensible Markup Language (XML) often used for web-based exchange of data and Resource Description Framework (RDF) used to represent information about resources in the Web. Although these formats make it possible to represent and exchange complex data, it is often difficult to query and analyze the data and data about the data (i.e., metadata).

In recent years many efforts have been put into developing data warehouse (DW) and business intelligence (BI) technologies. DW and BI technologies are very well-suited for storing and analyzing very large amounts of data. The data in a DW is prepared and stored in a way that makes analysis of it easy and efficient. It is therefore a natural step to apply DW technologies to web data and web metadata. But to apply DW technologies to complex web data is not straightforward and the DW community faces new challenges from the Web with its "always online paradigm" and the large amounts of data that must be handled. For example, it is nowadays often desired always to have fresh data, e.g., from a click-stream or from a Web-connected sensor, available in the DW with a very little delay. But to load huge amounts of data into a DW in (near-)real-time, instead of at regular intervals as traditionally done, is

challenging. Other challenges include to find solutions that allow easy and flexible exchange of DW data via the Web using XML and efficient extraction of RDF-based data to load into a DW.

The work that led to this thesis was mainly done in relation to the European Internet Accessibility Observatory (EIAO) project. The goal of the EIAO project is to build an observatory that automatically evaluates the *accessibility* of 10,000 European web sites every month. In other words, the projects considers how well web resources can be used by users with special needs such as a blind user who uses a screen reader. A web page that can also be used by such users is said to be *accessible*. The World Wide Web Consortium has published guidelines about how to make web resources accessible. For many of those guidelines, it can be automatically checked if web resources follow them. This is exactly what is done by the EIAO project. In the project, a crawler and tools and measures for evaluation of the accessibility of web resources are built and used. These accessibility evaluations result in large amounts of complex data. To make analysis of this data easy, fast, and reliable, it is collected in a DW called EIAO DW. To specify and develop this DW and supporting DW technologies for it has been the purpose of the Ph.D. project that led to this thesis. The supporting DW technologies are, however, designed to be general and can thus also be applied to DWs designed for other data than accessibility data.

To make it possible for everyone to use the developed solutions and to verify and understand the results, all software in the EIAO project is open source. This holds both for software used in the EIAO project and for software developed in the project. Thus, all the software presented in the remainder of this thesis is available as open source software.

Chapter 2 surveys the possibilities for using open source BI products as of End 2004 motivated by the fact that use of open source BI tools in industry is not common. First, the chapter presents some of the commonly used open source licenses. Then three Extract-Transform-Load (ETL) tools, three On-Line Analytical Processing (OLAP) servers, two OLAP clients, and four Database Management Systems (DBMSs) are considered in the survey. All the tools are evaluated against criteria relevant to the use of BI in industry. It is concluded that the DBMSs are the most mature of the tools and applicable to real-world projects, while the ETL tools are the least mature and in general not ready for use in industry.

Chapter 3 describes release 1 (from Mid 2006) of the EIAO DW used in the EIAO project. The chapter gives a brief introduction to the field of accessibility and to the entire architecture used in the EIAO project. The EIAO DW is a web warehouse built to make analysis of complex data about (in)accessibility of web resources easy, fast, and reliable. To do this, a general DW schema must be used and complex aggregation functions applied. It is believed that this work is the first to develop a general and

scalable BI solution to the field of accessibility. The conceptual, logical, and physical models are presented, as well as the RDF source data. The ETL procedure is also briefly presented. Bad performance when extracting the RDF based source data is a problem for the used solution.

Chapter 4 describes 3XL, a proposal for how to store very large Web Ontology Language (OWL) graphs efficiently. Motivated by our previous experiences with performance problems when storing and extracting large RDF data sets in general schemas, this chapter proposes a novel way to make a specialized schema for the data to store. To do this, 3XL focuses on the subset of RDF graphs that are also OWL graphs since they have some convenient characteristics that make it possible to optimize the schema. 3XL generates the specialized schema once and for all based on an OWL Lite ontology that describes the "structure" of the data to store. In contrast to a generic schema with few but large and narrow tables, 3XL has many wide tables, in particular at least one table for each OWL class. A theoretical analysis shows that 3XL can insert much fewer rows than a generic solution. In the used DBMS (PostgreSQL), the fewer rows result in much less storage overhead from rows. In the shown example with data about 10^7 instances, a specialized schema requires up to 8GB less storage than a generic schema. Further, the rows in the specialized schema are spread over more tables so it is faster to locate data.

Chapter 5 investigates automatic and effective bidirectional transfer between relational and XML data. This is motivated by the increasing exchange of relational data through XML based technologies such as web services. In such a use-case, data is exported from a relational database to XML from which the data must be importable into another relational database. Further, the XML may be updated such that only the resulting XML is present afterwards (and not a log of the changes). Based on the updated XML, it must the be possible to update the original database to reflect the changes to the XML. To set this up manually is cumbersome and a lot of hand-coding is needed. As a remedy to this situation the chapter presents RELAXML. With RELAXML the user must only specify what data to export and the structure of the XML. RELAXML does then take care of the export/import of data such that no custom-coding is needed. Before exporting, RELAXML automatically detects if all needed data is included to make it possible to re-import the data set and solves the problems or warns the user if any problems are detected. An implementation is presented and performance studies show that the suggested solution has a reasonable overhead compared to specialized, hand-coded solutions.

Chapter 6 considers regression test of ETL software. ETL software tends to be complex and error prone and may often be changed to increase performance or to handle changed data sources. Thus regression test is useful for ETL software. But traditionally this requires large manual efforts to set up. The chapter points out crucial differences between testing in "normal" software development and ETL development

and, based on these, the tool ETLDiff is presented. ETLDiff analyzes the DW schema and detects which parts of the data should not change between ETL runs on the same source data (some parts such as surrogate keys are allowed to change). Based on this analysis and optional user specification about what data to consider, ETLDiff compares test results to previous test results or other reference results and points out differences. When ETLDiff is used, a regression test can be set up in minutes instead of in days as when manual coding is done. The chapter presents a performance study of a prototype of ETLDiff. The results show that the running time scales linearly in the data size and that the solution is efficient enough to be used for regression testing on a desktop PC.

Chapter 7 investigates how to insert data into so-called right-time DWs. Traditionally, data has been bulk-loaded into DWs at regular intervals but recently it has become popular to insert new data as soon as it appears by using traditional SQL INSERT statements. This makes data available quickly, but performance suffers when the data amounts grow as when, e.g, click-streams are considered. There is thus a need to be able to make data available quickly while still preserving a high insert performance. The chapter presents the middleware system RiTE that provides such a solution which works transparently to both consumers and the producer. When RiTE is used, data can be inserted quickly by a producer and become available to consumers exactly when needed. As a remedy to obtain this, RiTE includes a novel main-memory based catalyst that provides fast storage. The movement of data between the different parts of the system and the DBMS can be controlled by user-defined policies that take the user's requirements for freshness, availability, and persistency into consideration. The chapter presents experiments that show that a prototype of RiTE provides INSERT-like data availability, but up to 10 times faster, i.e., with bulk-load speeds.

Appendix A presents the conceptual model for release 2.0 (from Mid 2007) of the EIAO DW. Compared to the conceptual model presented in Chapter 3, several changes have occurred to reflect the changed requirements and available data from the entire EIAO project.

The thesis is organized as a collection of individual papers. The chapters are organized such that material that is used in or motivates the work in another chapter appears first. But each chapter/appendix is self-contained and can be read in isolation. The chapters have been slightly modified during the integration such that, for example, their bibliographies have been combined to one and references to "this paper" have been changed to references to "this chapter". There are some overlaps between the descriptions of the conceptual model for EIAO DW releases 1 and 2 given in Chapter 3 and Appendix A, respectively. Concretely Appendix A describes

conceptual classes that are also described in Chapter 3 but Appendix A also describes new classes from release 2.

The papers included in the thesis are listed below. Chapter 2 is based on Paper 1, Chapter 3 is based on Paper 2 and so on until Chapter 7 which is based on Paper 6. Appendix A is a shortened version of Paper 7.

1. C. Thomsen and T. B. Pedersen. A Survey of Open Source Tools for Business Intelligence. In *Proceedings of the 7th International Conference on Data Warehousing and Knowledge Discovery*, pp. 74–84, 2005.

2. C. Thomsen and T. B. Pedersen. Building a Web Warehouse for Accessibility Data. In *Proceedings of the 9th ACM international workshop on Data warehousing and OLAP*, pp. 43–50, 2006.

3. C. Thomsen and T. B. Pedersen. 3XL: Efficient Storage for Very Large OWL Graphs. *In preparation for submission*, 18 pages, 2007.

4. S. U. Knudsen, T. B. Pedersen, C. Thomsen, and K. Torp. RELAXML: Bidirectional Transfer between Relational and XML Data. In *Proceedings of the 9th International Database Engineering & Application Symposium*, pp. 151–162, 2005.

5. C. Thomsen and T. B. Pedersen. ETLDiff: A Semi-automatic Framework for Regression Test of ETL Software. In *Proceedings of the 8th International Conference on Data Warehousing and Knowledge Discovery*, pp. 1–12, 2006.

6. C. Thomsen, T. B. Pedersen, and W. Lehner. RiTE: Providing On-Demand Data for Right-Time Data Warehousing. To appear in *Proceedings of the 24th International Conference on Data Engineering*, 10 pages, 2008.

7. T. B. Pedersen and C. Thomsen. *EIAO Deliverable 6.1.1.1-2, Appendix A: Conceptual Model for EIAO DW, R2*, 27 pages, 2007.

Chapter 2

A Survey of Open Source Tools for Business Intelligence

The industrial use of open source Business Intelligence (BI) tools is not yet common. It is therefore of interest to explore which possibilities are available for open source BI and compare the tools.

In this survey chapter, we consider the capabilities of a number of open source tools for BI. In the chapter, we consider three Extract-Transform-Load (ETL) tools, three On-Line Analytical Processing (OLAP) servers, two OLAP clients, and four database management systems (DBMSs). Further, we describe the licenses that the products are released under.

It is argued that the ETL tools are still not very mature for use in industry while the DBMSs are mature and applicable to real-world projects. The OLAP servers and clients are not as powerful as commercial solutions but may be useful in less demanding projects.

2.1 Introduction

The use of Business Intelligence tools is popular in industry [76, 85, 86]. However, the use of open source tools seems to be limited. The dominating tools are closed source and commercial (see for example [76] for different vendors' market shares for OLAP servers). Only for database management systems (DBMSs), there seems to be a market where open source products are used in industry, including business-critical systems such as online travel booking, management of subscriber inventories for tele communications, etc. [72]. Thus, the situation is quite different from, for

example, the web server market where open source tools as Linux and Apache are very popular [114].

To understand the limited use of open source BI tools better, it is of interest to consider which tools are available and what they are capable of. This is the purpose of this chapter. In the European Internet Accessibility Observatory (EIAO) project, where accesibility data is collected, it is intended to build a BI solution based on open source software. It is therefore of relevance for this project to investigate the available products.

In the survey we will consider products for making a complete solution with an Extract-Transform-Load (ETL) tool that loads data into a database managed by a DBMS. On top of the DBMS, an On-Line Analytical Processing (OLAP) server providing for fast aggregate queries will be running. The user will be communicating with the OLAP server by means of an OLAP client. We limit ourselves to these kinds of tools and do not consider, for example, data mining tools or Enterprise Application Integration (EAI) tools. Use of data mining tools would also be of relevance in many BI settings, but data mining is a more advanced feature which should be considered in future work. EAI tools may have some similarities with ETL tools, but are more often used in online transactional processing (OLTP) systems.

The rest of the chapter is structured as follows. Section 2.2 gives a primer on open source licenses. Section 2.3 presents the criteria used in the evaluation of the different tools. Section 2.4 considers ETL tools. Section 2.5 deals with OLAP servers, while Section 2.6 deals with OLAP clients. Finally, we consider DBMSs in Section 2.7 before concluding and pointing to future work in Section 2.8.

2.2 Open Source Licenses

To make the findings on licenses more comprehensible, we include a description of the open source licenses that will be referred to later in the chapter. The *GNU General Public License (GPL)* [43] is a classic, often used open source license. Any user is free to make changes to the source code. If the changed version is only used privately, it is not a requirement that its source code is released. If it, however, is distributed in some way, then the source code must be made available under the GPL (i.e. also released as open source that any user is free to change and copy). It should be noted that a library released under the GPL will require any program that uses it to be licensed under the GPL. This is not the case when using *GNU Library General Public License (LGPL)* [44] which apart from that is much like the GPL. The *Common Public License (CPL)* [27] was developed by IBM as an open source license. Like the GPL, the CPL requires that the source code for a modified version of a program is made publicly available if the new version is distributed to anyone.

Programs and libraries released under the CPL may be used from and integrated with software released under other (also closed source) licenses.

The *Mozilla Public License* [70] is also an open source license that requires the code for any distributed modified works to be made publicly available. It is allowed to use a library under the Mozilla Public License from a closed source application. Thus the license has some similarities with the LGPL. The *Apache License* [6, 7] allows the code to be used both in open source, free programs and in commercial programs. It is also possible to modify the code and redistribute it under another license under certain conditions (e.g. the use of the original code should be acknowledged). Version 1.0 and 1.1 of the Apache License [6] included requirements about the use of the name "Apache" in documentation and advertising materials. That meant that the license should be modified for use in non-Apache projects. This was changed with version 2.0 [7]. The *BSD License* [19] is a very liberal open source license. It is permitted to use source code from a BSD licensed program in a commercial, closed source application. As long as any copyright notices remain in the modified code, there are no requirements saying that modifications of the code should be BSD licensed or open source.

2.3 Conduct of the Survey

In this section, we present the criteria used for the evaluation of the considered products. The criteria with a technical nature have been inspired by the functionality offered by the leading commercial BI tools. Other criteria, such as the type of license, are interesting when looking at open source tools. Based on the criteria given below, we collected data about the products (all found on the Internet) by examining their source code, available manuals, and homepages including any forums. The findings were collected in November-December 2004.

Criteria for All Categories When deciding between different products, a potential user would most often prefer a product that is compatible with his[1] existing operating system and hardware. Thus, for all the products, it is of interest to investigate which hardware and software platforms the tools are available for. In this survey we will only look at open source products. As described in Section 2.2 there are, however, many different open source licenses that have different permissions and restrictions. Therefore, the license used by a product is also of great interest.

Criteria for ETL Tools When comparing ETL tools, there are several criteria to consider. First, it should be considered which data sources and targets a given tool

[1] We use "his" as short for "his/her".

supports. Here, it should be considered whether the tool is for loading data into ROLAP or MOLAP systems, i.e. into relational tables or multidimensional cubes [86]. In many practical applications, it should be possible to extract data from different sources and combine the data in different ways. Further, it should be possible to load the data into different tables/cubes. Therefore support for these issues should be considered. It should also be considered which types of data sources an ETL tool can extract data from and whether it supports incremental load in an automatic fashion (not requiring two separate flows to be specified). It is also of interest how the ETL process is specified by the user, i.e whether a graphical user interface (GUI) exists and if the user can specify the process directly by means of some specification language. Another important issue for ETL tools is their capabilities for data cleansing. Here it should be considered how data cleansing is supported, i.e. if predefined methods exist and how the user can specify his own rules for data cleansing.

Criteria for OLAP Servers For an OLAP server it is of interest to know how it handles data. It should thus be considered whether the tool is ROLAP, MOLAP, or HOLAP oriented, where HOLAP is short for Hybrid OLAP [86]. Further, it is of interest if the product is capable of handling large data sets (for example, data sets greater than 10 gigabytes). It should also be taken into account whether an OLAP server has to be used with a specific DBMS or if it is independent of the underlying DBMS. Precomputed aggregates can in many situations lead to significant performance gains. It is therefore relevant to see whether an OLAP server can use aggregates and if so, whether the user can specify which aggregates to use. Finally, it is of relevance to investigate which application programming interfaces (APIs) and query languages an OLAP server supports. A product that uses standards or de-facto standards is much more useful with other tools than a product using a non-standard API or query language.

Criteria for OLAP Clients For an OLAP client it should be considered which OLAP server(s) the OLAP client can be used with. As for OLAP servers, it should also be taken into account which API(s) and query language(s) the OLAP client supports. With respect to reports, it is interesting to see if the OLAP client supports prescheduled reports, perhaps through a server component. If so, the user could, for example, make the OLAP client generate a sales report every Friday afternoon. When a report has been generated (manually or as prescheduled report), it is often useful to be able to export the report to some common format that could be emailed to someone else. Therefore, it should be investigated which export facilities an OLAP client offers. In generated reports, different types of graphs are often used. It should thus also be considered how well an OLAP clients supports different kinds of graphs.

Criteria for DBMSs There are many possible criteria to consider for DBMSs. In this survey we will, however, only look at criteria directly relevant for BI purposes. First of all, a DBMS should be capable of handling large data sets if the DBMS is to be used in BI applications. Thus this is an issue to investigate. When choosing a DBMS for BI, it is also of relevance which performance improving features the DBMS offers. In this survey we will look into the support for materialized views that can yield significant performance gains for precomputed aggregates. Many commercial ROLAP systems use bitmap indices to achieve good performance [86]. It is also of interest to find out whether these are supported in the considered products. Further, in a typical schema for a data warehouse, star joins may be a faster to use and, thus, the support for these is an issue. Finally, we will consider partitioning which can yield performance improvements and replication which may improve performance and reliability.

2.4 ETL Tools

In this section, we will consider the three ETL tools Bee, CloverETL, and Octopus. These were all the available tools we found. We found many other open source ETL projects that were not carrying any implementation, but more or less only stated objectives. Examples of such projects are OpenSrcETL [80] and OpenETL [77]. Another disregarded project, was cplusql [28] which had some source code availble, but for which we did not find any other information.

Bee Bee version 1.1.0 [8] is a package consisting of an ETL tool, an OLAP server, and an OLAP client web interface. The ETL tool and the OLAP server of Bee are ROLAP oriented. Bee is available under both an open source GPL license and a commercial license. Bee is implemented mainly in Perl with parts implemented in C. Therefore, the access to data is provided by the Perl module DBI. Bee comes with its own driver for comma-separated files. To extract data, Bee needs a small server application (included) to be running on the host holding the data. Bee is primarily written for Linux, but is also running on Windows platforms. The mentioned server application needed for extracting data runs on different varieties of UNIX and Windows. The ETL process can be specified by means of a GUI. The GUI will create an XML file defining the process. It is thus also possible for the user to use Bee without using the GUI by creating the XML file manually. It is possible to have several flows and combine them. It is also possible to insert into more than one table in the database. There seems to be no support for automatic incremental loading. The possibility for data cleansing is introduced by means of allowing the user to write custom transformations in Perl. A standard library of transformations is not included. Thus the user needs to program any needed transformation.

CloverETL CloverETL version 1.1.2 [24] is also a ROLAP oriented ETL tool. Parts of it are distributed under the GPL license whereas other parts are distributed under the LGPL license. CloverETL is implemented in Java and uses JDBC to transfer data. The ETL process is specified in an XML file. In this XML file, a directed graph representing the flow must be described. Currently, CloverETL does not include a GUI, but work is in progress with respect to this. CloverETL supports combination of several flows as well as import to several tables in the database. There is no support for automatic incremental load. With respect to cleansing, CloverETL supports insertion of a default value, but apart from this, the user will have to implement his own transformations in Java.

Octopus Octopus version 3.0.1 [34] is a ROLAP oriented ETL tool under the LGPL license. It is implemented in Java and is capable of transferring data between JDBC sources. Octopus is bundled with JDBC drivers for XML and comma-separated files. Further, it is possible to make Octopus create SQL files with insert and DDL statements that can be used for creating a database holding the considered data. Like Bee, Octopus is shipped with a GUI that creates an XML file specifying the ETL process. Octopus can also be used without the GUI and as a library. Octopus is created for transferring data between one JDBC source and another. It is apparently not possible to combine data from one database with data extracted from another. It is possible to extract data from more than one table in the same database as well as insert into more than one table in the target database. There is no direct support for automatic incremental loading. Basic data cleansing functionality is provided. It is possible to make Octopus insert a default value, shorten too long strings, replace invalid foreign key values, find and replace values, do numeric conversions, and change date formats. These cleansings are done by predefined transformations. The user can also implement transformations on his own in Java and JavaScript.

General Comments The considered open source ETL tools are still not as powerful as one could wish. For example, most of the data cleansing to be done must be coded by the user (with the exception of Octopus which provides some default transformations for very basic data cleansing). Further, the products do not support automatic incremental load which would be very useful for everyday use of the products. In general, the quality of the documentation for the described products is not very good or comprehensive. Further, not much documentation is available. An exception is again Octopus for which a manual of more than 120 pages is available. However, also this manual is not complete. For example, it explains how to set up which *logger* to use but does not tell about the differences between the available loggers. Thus the quality of the open source ETL products is still not as high as the quality of many commercially available products. Indeed it would probably be diffi-

cult to use one of the open source ETL tools for a demanding load job in an enterprise data warehouse environment.

2.5 OLAP Servers

In this section, we will consider the three OLAP servers Bee, Lemur, and Mondrian. Another possible candidate for consideration would be pocOLAP [93] which, however, in the documentation is said not to be an OLAP server. It provides access to data from a DBMS through a web-interface but is not intended to provide advanced OLAP functionality or real-time data analysis. OpenRolap [81] is a related tool which generates aggregate tables for a given database. Apart from these tools we did not find any candidates. As for the ETL tool category, there exist other projects that currently carry no code, but only state objectives. An example of such a project is gnuOLAP [45].

Bee The OLAP server of the Bee package is, as previously stated, a ROLAP oriented server. It uses a MySQL system to manage the underlying database and aims to be able to handle up to 50GB of data efficiently [9]. Despite this, it does not seem to be possible to choose which precomputed aggregates to use. From the documentation, it is not clear which query language(s) and API(s) Bee supports. In general, there is not much English documentation available for Bee, neither from the homepage [8], nor in the downloadables.

Lemur Unlike the other OLAP servers considered in this chapter, Lemur [62] is a HOLAP oriented OLAP server. It is released under the GPL license and is written in C++ for Linux platforms, but is portable. Lemur is a product under development and still has no version number. The homepage for the Lemur project [62] states that for now, the primary goal is to support the developers research interests and that their goals are believed to be too ambitous to deliver usable code now. This is also reflected in the fact that the API is still being designed and in reality is not available for use from outside the Lemur package. Further the user would need to implement methods to load data from a database on his own. It is also not possible to specify the aggregates to be used. No information on how well Lemur scales when applied to large data sets has been found. In summary, the Lemur project is not of much practical use for industry projects so far. However, the goal of eventually producing a HOLAP oriented server outperforming Mondrian (see below) is interesting.

Mondrian Mondrian 1.0.1 [67] is an OLAP server implemented in Java. It is ROLAP oriented and can, unlike Bee, be used with any DBMS for which a JDBC driver

exists. Mondrian is released under the CPL license [27]. The current version of Mondrian has an API that is similar to ADO MD from Microsoft [68]. Support for the standard APIs JOLAP [51] and XMLA [124] is planned. Further, the MDX query language [106], known from Microsoft's products, is supported by Mondrian. By default Mondrian will use some main memory for caching results of aggregation queries. It is, however, neither possible for the user to specify what should be cached nor which aggregates should exist in the database. The documentation states that Mondrian will be able to handle large data sets if the underlying DBMS is, since all aggregation is done by the DBMS.

General Comments The Mondrian OLAP server seems to be the best of the described products. Lemur is for the time being not usable for real world applications while it is difficult to judge Bee because of its lack of English documentation. Mondrian is, however, a usable product which works with JDBC-enabled DBMSs. For none of the products, it seems possible to choose which aggregates to use. In most environments this feature would result in significant performance improvements.

2.6 OLAP Clients

In this section, we will describe the OLAP clients Bee and JPivot. These were the found open source OLAP clients that are actually implemented.

Bee The Bee project also provides an OLAP client. The client is web-based and is used with the Bee OLAP server. Currently, Microsoft Internet Explorer and Mozilla browsers are explicitly supported in the downloadable code. Again, it has not been possible to determine which API(s) and query language(s) Bee supports. The Bee OLAP client can interactively present multidimensional data by means of Virtual Reality Modeling Language (VRML) technology [112]. Bee can generate different types of graphs (pie, bar, chart, etc.) in both 2D and 3D. It is possible to export data from Bee to Excel, Portable Document Format (PDF), Portable Networks Graphics (PNG), PowerPoint, text, and Extensible Markup Language (XML) formats. Connection with the statistical package R [95] is also evaluated. It does not seem to be possible to preschedule reports.

JPivot JPivot version 1.2.0 [54] is a web-based OLAP client for use with the Mondrian OLAP server. However, the architecture should allow for later development of a layer for XMLA [124]. As Mondrian, JPivot uses MDX as its query language. It is written in Java and JSP. JPivot generates graphs by means of JFreeChart [53] which provides different kinds of 2D and 3D graphs. With respect to export of reports,

JPivot is limited to Portable Document Format (PDF) and Excel format. Support for prescheduled reports has not been found. JPivot is released under a license much like the Apache Software License Version 1.1 [6] (but without restrictions regarding the use of the name "Apache"). However, other software packages are distributed with JPivot and have other software licenses, e.g. JFreeChart which uses the LGPL license.

General Comments Both the considered OLAP clients are to be used with specific OLAP servers, namely Bee with Bee and JPivot with Mondrian. Both of them are web-based such that specific software does not have to be installed at client machines already equipped with a browser. Both products are capable of exporting generated reports to other commonly used file formats such as PDF, but neither of them supports prescheduled reports.

2.7 DBMSs

In this section we consider four open source DBMSs: MonetDB, MySQL, MaxDB, and PostgreSQL. Other open source DBMSs are available, but these four were chosen as they are the most visible, well-known high-performance DBMSs.

MonetDB MonetDB, currently in version 4.4.2, is developed as a research project at CWI. MonetDB is "designed to provide high performance on complex queries against large databases, e.g. combining tables with hundreds of columns and multi-million rows" [69]. To be efficient, MonetDB is, among other techniques, exploiting CPU caches and full vertical fragmentation (however, the fragments must be placed on the same disk). It thus uses very modern and often hardware-near approaches to be fast. MonetDB is mainly implemented in C with some parts in C++. It is available for 32- and 64-bit versions of Linux, Windows, MacOS X, Sun Solaris, IBM AIX, and SGI IRIX. MonetDB comes with a license like the Mozilla Public License (references to "Mozilla" are replaced by references to "MonetDB") [69]. With respect to features often usable in a BI context, it is interesting to notice that MonetDB does not support bitmap indices, materialized views (normal views are supported), replication, or star joins. However, this does not mean that MonetDB is not usable for BI purposes. On the contrary, MonetDB has been successfully applied in different BI contexts [69]. Currently, the developers are working on improving the scalability for OLAP and data mining in the 64-bit versions of MonetDB.

MySQL MySQL is a very popular open source database with more than five millions installations [73]. The latest production release is version 4.1, and version 5.0

is in the alpha stage. MySQL is implemented in C and C++ and is available for a large variety of 32- and 64-bits platforms. Users of MySQL can choose between an open source GPL license and a commercial license that gives permissions not given by the GPL license. For BI purposes, MySQL lacks support of materialized views (even ordinary views are not available until version 5.0), bitmap indices and star joins. However, one-way replication (i.e. one master, several slaves) is supported and partitioning is to some degree supported by the *NDB Cluster* (NDB is a name, not an acronym) on some of the supported platforms. Further, MySQL is capable of handling data sets with terabytes of data as documented in case studies available from [73].

MaxDB MaxDB [64] version 7.5 is another RDBMS distributed by the company MySQL AB which also develops MySQL. Formerly, MaxDB was known as SAP DB (developed by SAP AG). MaxDB is developed to be used for OLTP and OLAP in demanding environments with thousands of simultanous users. It is implemented in C and C++ and is available for most major hardware platforms and operating system environments. As MySQL, it is licensed under two licenses such that users can choose between an open source license (GPL) or a commercial. MaxDB is designed to scale to databases in the terabyte sizes, but there is no user controlled partitioning. It is, however, possible to specify several physical locations for storage of data, and MaxDB will then automatically divide table data between these partitions. There is no support for materialized views (ordinary views are supported), bitmap indexes, or star joins. MaxDB supports one-way replication, also with MySQL such that either of them can be the master.

PostgreSQL PostgreSQL [94] is also a very popular open source DBMS. At the time of this writing, version 8.0 is just about to be released. PostgreSQL is implemented in C and has traditionally only been available for UNIX platforms. From version 8.0, Windows is, however, natively supported. Originally, PostgreSQL is based on the POSTGRES system [107] from Berkeley and has kept using a BSD license. PostgreSQL supports large data sets (installations larger than 32 terabytes exist) and one-way replication. A multiway solution for replication is planned. There is no support for partitioning, bitmap indices or materialized views (ordinary non-materialized views are supported). However, materializations of views may be done in PostgreSQL by using handcoded triggers and procedures [42]. Further, in a research project at North Carolina State University, materialized views are integrated into a derived version of PostgreSQL [101]. Bitmap indices are planned to be supported in a future release.

General Comments The considered DBMSs have different strengths and weaknesses, and so there is not a single of them to be chosen as *the best*. In general, these open source products support more advanced features such as partitioning and replication. Further, the DBMSs are capable of handling very large data sets, are available for a number of platforms, and are very reliable. The category of DBMSs is thus the most *mature* of the considered categories.

2.8 Conclusion and Future Work

Of the considered categories of open source tools (ETL tools, OLAP clients, OLAP servers, and DBMSs), DBMSs are the most mature. They offer advanced features and are applicable to real-world situations where large data sets must be handled with good performance. The ETL tools are the least mature. They do still not offer nearly the same functionality as proprietary products. With respect to the OLAP servers, there is a great difference in their maturity. A product like Lemur is still very immature, while a product like Mondrian is usable in real-world settings. However, important features, such as the opportunity to specify which aggregates to use, are still missing. The OLAP clients are also usable in practical applications. However, they are not very general and can only be used with specific OLAP servers. In general, one of the largest problems for many of the tools is the lack of proper documentation, often making it very difficult to decide how a specific task is performed in a given product.

If one were to create a complete BI installation with open source tools, it would probably be created with JPivot and Mondrian as OLAP client and server, respectively. Which one of the DBMSs should be used would depend on the situation. The ETL tool would then probably be CloverETL, if one did not handcode a specialized tool for the installation. In many BI installations, data mining solutions would also be interesting to apply. The available open source data mining applications should therefore be explored in future work.

Chapter 3

Building a Web Warehouse for Accessibility Data

As more and more information is available on the Web, it is a problem that many web resources are not *accessible*, i.e., are not usable for users with special needs. For example, for a web page to be accessible, it should give text alternatives (i.e., explanatory texts) for images such that blind users that have the web pages read aloud automatically also can obtain information about the images. In the *European Internet Accessibility Observatory* (EIAO) project, a crawler that will evaluate the accessibility of thousands of European web sites is built. The crawler frequently performs many tests of the web sites and thus very large amounts of accessibility data are generated. Based on open-source software, a data warehouse (DW) called *EIAO DW* is built to make analysis of the complex accessibility data easy, reliable and fast. The EIAO DW is, thus, a data warehouse which measures *properties of the Web* or, in other words, a *web warehouse*. It is believed that this work is the first to address the application of business intelligence (BI) techniques to the complex field of accessibility in a general and scalable way. This chapter describes how the EIAO DW is designed and built. The chapter introduces accessibility and the EIAO project to give a background for the design of EIAO DW. Then, the conceptual, logical and physical models are presented. The chapter also gives descriptions of the complex Resource Description Framework (RDF) source data and complex accessibility aggregation functions supported by EIAO DW.

3.1 Introduction

To be able to use and access the Web is getting increasingly important. In Denmark, all people employed by the state have access to their payslips only via the Web. In many countries, it is the case that nearly all authorities, companies, and organizations have web sites. For some of these, such as web shops, the web site is their primary means for communication with the customers or users. For many web sites it is, however, a problem that they are not *accessible*, i.e., are not usable, for users with special needs, for example a blind user using a so-called *screen reader* which is a program that reads text aloud. Consider as an example a web page with images that are used as links. The images contain arrows and the texts "Continue" and "Go back", respectively. For a blind user, it is necessary that text alternatives (i.e., explanatory texts) are available for the images such that the text alternatives can be read aloud by the screen reader. If text alternatives are not present, the blind user cannot obtain the information needed to navigate on the page (or will, at the best, have to guess how to do this based on file names etc.)

It is possible to do automatic testing for some of the constructs that make web pages inaccessible. To do such testing systematically and store the results for thousands of web sites generates very large amounts of data. In this chapter, we describe the web warehouse *EIAO DW* which is a data warehouse (DW) that measures the web with respect to accessibility. The EIAO DW will hold detailed data about (in)accessibility of thousands of web sites and offers convenient, flexible and reliable online access to the collected accessibility data. Both historical and current data is available in the EIAO DW in order to facilitate trend analysis, mining of interesting patterns, etc. Further, all components in the EIAO DW are based on open-source technology. This introduces some challenges compared to if commercial solutions were used.

Previously, other studies on evaluating web accessibility have been done. A thorough treatment is given in [125]. A survey of accessibility studies with focus on online resources for higher education is available from [100]. Watchfire WebXACT [113] is an example of an online accessibility evaluation service. For this service, the user, however, gives a URL and gets information about what to consider, i.e., the service is primarily for web developers. In the EIAO project, sites are monitored on a monthly basis and the data is put in a DW such that historical data is also available. The EIAO project can provide (aggregated) information to end users about accessibility of (groups of) web sites.

To the best of our knowledge, this work is the first to develop a general and scalable business intelligence (BI) solution to the field of accessibility. A very limited prototype for a crawler evaluating accessibility and storing the data in a simple DW has previously been described [105]. This prototype was implemented such that only

one specific issue regarding accessibility was measured and the DW schema was designed to store data only for this issue. The EIAO DW described in the current chapter is much more general. It measures dozens of different accessibility issues and more can be added without changes to the DW schema. Further, it handles much more complex data and aggregation functions. Thus, the EIAO DW is believed to be the first DW to deal with accessibility data in a truly general and scalable manner.

The field of accessibility is a complex field and makes it challenging to make data models for the DW. Recently, much work has gone into bringing the Web and data warehousing together. Pérez *et al.* [90] give a survey of the most relevant techniques. A prominent example on using DWs to treat data from web logs is given by Kimball and Merz [56]. In the WHOWEDA project, a data warehouse that holds web data is built. A so-called *web schema* that holds information on structure, metadata, content, link structure is automatically generated [11]. The WHOWEDA project stores data from the Web (e.g., the content of a web page) in a warehouse. In contrast, what we describe in the current chapter, is a DW that stores (accessibility) data *about* web resources, but not the web resources themselves.

The rest of the chapter is organized as follows. In the next section, we give a short introduction to accessibility and a description of the European Internet Accessibility Observatory (EIAO) project for which the EIAO DW is built. The conceptual model for EIAO DW is described in Section 3.3. Section 3.4 describes the logical model while Section 3.5 describes the physical model. In Section 3.6, the source data and Extract–Transform–Load (ETL) process are described. The accessibility aggregation functions implemented by EIAO DW are described in Section 3.7. Section 3.8 concludes and points to future work.

3.2 Web Accessibility

3.2.1 Accessibility of Web Resources

To give everybody equal opportunities it is important to make web resources *accessible*. For a web resource to be accessible means that people with disabilities can also use the web resource. Thus, accessibility can considered from many different contexts. For example, a web page should be understandable for a blind user using a screen reader or a so-called *Braille display* that raises small pins physically such that the user can feel the letters with his finger. Another example is that it should be possible to use a web resource without using a pointing device such that a physically disabled user can also use the resource.

As part of the Web Accessibility Initiative (WAI) [121], some guidelines called Web Content Accessibility Guidelines (WCAG) have been defined [122, 123] by the World Wide Web Consortium (W3C). If followed, the WCAG guidelines will in-

crease accessibility; both for disabled users and for other users such as people using a hand-held device with a small screen. The guidelines, for example, state that a web page should provide text alternatives for all non-text content. If text alternatives are available for all non-text content, a blind user will also be able to use an image for navigation purposes and a deaf user will also be able to use a web page which uses sound. The current version of WCAG is version 1.0 [122], but soon version 2.0 [123] will be available as a W3C Recommendation.

Many of the WCAG recommendations may be checked automatically. For example, it is easy to check if an XHTML `image` element has an `alt` attribute with a text alternative. It is, however, not clear how to test if the text is meaningful and indeed is an alternative.

To have an accessible design is not only advantageous for the end user. For a commercial web site, it is important to make sure that the web site's visitors actually can and will use the site. If, for example, text alternatives are not present for images used for navigation, a blind user may not be able to navigate the web site and may give up and instead go to a competitor's (accessible) web site. Also for governmental sites it is important. Both to give everybody equal opportunities but also to make e-government and self-service a success. To do this, it is a necessity that the offered web resources are accessible such that everybody can use them. Today, accessibility is recognized as an important field and public institutions have policies about accessibility and work actively on improving the level of accessibility on the web. For example, the European Union has an *eInclusion* program and has adopted the WCAG guidelines [35, 36].

3.2.2 The EIAO Project

The European Internet Accessibility Observatory (EIAO) project [37], will develop large-scale accessibility benchmarking. In the EIAO project, the accessibility of 10,000 European web sites will be monitored automatically. The found accessibility data will be stored in the data warehouse EIAO DW. The accessibility data will be frequently updated and will be available online. In this section, we describe release 1 of the software developed by the EIAO project. This release is currently being tested with monitoring of about 150 sites. Release 2 which will be used for monitoring 10,000 sites is currently being developed. Release 2 will handle around 200 million new facts each month.

Note that all software products used and developed by the EIAO project are open source. This introduces some challenges compared to if commercial products were used. For example, the used open source database management system (DBMS) does not support materialized views as market-leading commercial DBMSs do.

The entire architecture for the observatory is outlined in Figure 3.1. A central URL repository holds a list of domains to collect data from. A crawler fetches URLs

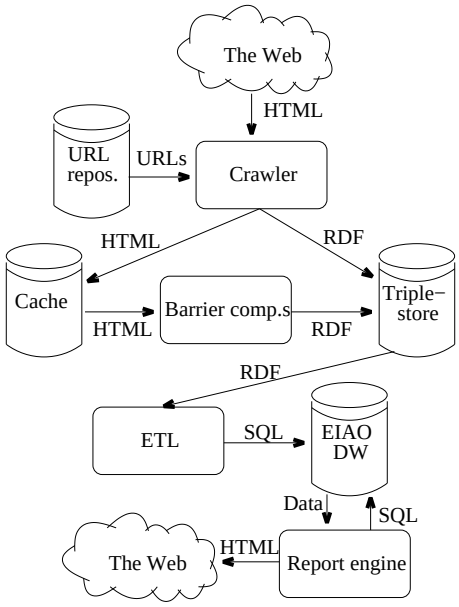


Figure 3.1: The architecture for the EIAO observatory

from the repository and starts sampling pages from each of the web sites. To save time, storage, and bandwidth, the crawler only samples a part of a given web site. The crawler downloads a number of pages from the site by following links in a random manner. This is done by simulating a use scenario such that there is a probability p for following a link to read on another page and a probability $1 - p$ for not following the link and continue reading on the current page. Note that the entire HTML documents are downloaded by the crawler under all circumstances. The downloaded pages are stored in a cache and then evaluated by the so-called *barrier computations* explained below.

To evaluate the accessibility of a web page, a number of *Web Accessibility Metrics* (WAMs) have been defined [91]. These are rules that specify how to make statements about accessibility barriers of web resources. Different kinds of WAMs exist: Analytic WAMs (A-WAMs) analyze specific page elements (e.g., `image` elements in an HTML document to see if an `alt` attribute is present). The results from the A-WAMs are used by *barrier computations*. A barrier computation is a single function that uses data from the basic A-WAMs to detect if a barrier may be present. A *Barrier reporting WAM* (B-WAM) is a group of related barrier computations. Finally, *Composing WAMs* (C-WAMs) are functions that calculate the probability that a barrier is present. The result of a C-WAM is calculated by considering the results of the barrier computations for tested elements within the areas that the simulated user read.

The barrier computations deliver their results in the Evaluation and Report Language (EARL) [120] which is a Resource Description Framework (RDF) language [119]. Also, certain data from the crawler, such as information from the HTTP header, is in RDF. The RDF data is stored in a 3store [48] triplestore as it is being generated.

The relevant parts of the data from the triplestore is periodically loaded into the EIAO DW data warehouse by an ETL application. The data warehouse is implemented as a relational database in PostgreSQL 8.1 [94]. A graphical user interface (GUI) is implemented on top of the DW. The GUI is available via the Web and can present different reports. A user can choose to see aggregated results for a specific web site, for geographical regions and for sectors (e.g., "Radio stations" or "Banks").

Preliminary results indicate that on average 76 pages are downloaded from each considered web site. These pages on average have 247 different test results about accessibility of the elements on the page. Thus, if 10,000 sites are evaluated on a monthly basis, 187.7 millions test results will be generated each month. Clearly, it is a challenge to handle such data amounts efficiently.

3.3 Conceptual Model

In this section, we describe the conceptual model for EIAO DW. The notation is in UML [78]. Thus classes are represented as boxes with the class name on the top row. Below the class name follow the attributes of the class. Associations are represented by a line between the involved classes. In the conceptual model for EIAO DW, we have 42 classes with a total of 113 attributes. All the classes have ID attributes that are surrogate keys. Foreign keys referencing other classes have not been added as attributes, but are represented as associations. Some of the 42 classes will in the logical model be combined into nine different dimension tables. The classes for the fact table and association classes (which become so-called *bridge tables* in the dimensional design of the logical model) will not be in a dimension. Here we describe the classes grouped by the dimension they belong to in the logical model. The purpose and important characteristics of each class are described but due to space limitations each of the 113 attributes is not described individually. A thorough description of each class and attribute can be found in [89].

Result Dimension In the Result dimension, shown in Figure 3.2(a), there are two classes: The *Result* class and the *ResultType* class. The *Result* class is used to represent the outcomes of the barrier computations. Each barrier computation has a unique fail description that describes failing tests, but all barrier computations share the result "Test passed" for passed tests and "Unknown result" for cases where a barrier computation for some reason could not provide an answer. This also reveals

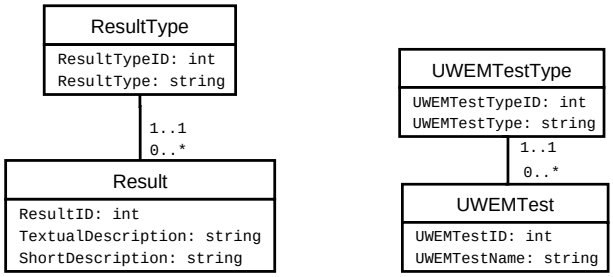


Figure 3.2: The (a) Result and (b) UWEMTest dimensions

that there are different types of results, namely *pass* results, *fail* results and *unknown* results. These are represented by *ResultType*. A result is associated with exactly one result type, but many results are associated with the fail result type. Note that *ResultType* only has an ID attribute and an attribute named ResultType. The purpose of having such a class without further attributes is to model the hierarchy, i.e., what *levels* are present.

DisabilityGroup Dimension In the DisabilityGroup dimension there is one class: *DisabilityGroup*. This class is used to represent different disability groups, e.g., people with functional blindness or people with physical disabilities. In this way it is possible only to consider the accessibility problems related to a specific group. There is also a general group for which all accessibility issues are considered. Apart from the ID attribute, there is only one attribute, DisabilityGroup, which holds the name of the represented group.

UWEMTest Dimension In the UWEMTest dimension, see Figure 3.2(b), there are two classes: *UWEMTest* and *UWEMTestType*. The class *UWEMTest* represents tests that have been defined by the Unified Web Evaluation Methodology (UWEM) [39] which the EIAO project has also been involved in. Currently, there are UWEM test types that deal with Cascading Style Sheets (CSS) and UWEM test types that deal with HTML. The types are represented by the *UWEMTestType* class.

BarrierComputationVersion Dimension In the BarrierComputationVersion dimension, shown in Figure 3.3, there are six classes: *BarrierComputationVersion*, *BarrierComputation*, *WAM*, *WCAGMinor*, *WCAGMajor*, and *WCAGType*. Different versions of the barrier computations may be implemented, e.g., if bugs are found and fixed and thus it should be possible to represent the versions. In case of a bug, it is easy to find the results that cannot be trusted and have to be updated, when it is

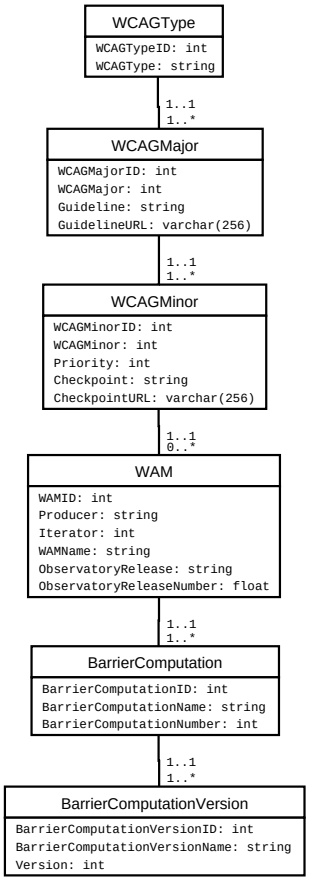


Figure 3.3: The BarrierComputationVersion dimension

known which version computed a given result. Thus versions are represented by the *BarrierComputationVersion* class. It should also be possible to represent the different generic barrier computations, i.e., the specification part of a barrier computation that does not change between different implementations. This is the purpose of the *BarrierComputation* class. Obviously, a barrier computation version is a version of exactly one barrier computation. On the other hand there might be many versions of one barrier computation. Thus, there is a many-to-one association from *BarrierComputationVersion* to *BarrierComputation.* A barrier computation is, as previously described, a single function out of possibly many in one B-WAM. B-WAMs are represented by the class *WAM* and there is a many-to-one association from *BarrierComputation* to *WAM.* A (B-)WAM is considering exactly one WCAG checkpoint, represented by *WCAGMinor,* and there is a many-to-one association from *WAM* to *WCAGMinor.* A WCAG checkpoint belongs to a WCAG guideline. The WCAG

guidelines are represented by *WCAGMajor* and there is a many-to-one association from *WCAGMinor* to *WCAGMajor*. Currently, all the used WCAG checkpoints and guidelines are from WCAG 1.0. To prepare for future use of WCAG 2.0, there is also a class *WCAGType* to represent the different WCAG versions.

The class *BarrierComputation* is also associated with the class *DisabilityGroup* in the DisabilityGroup dimension and with the class *UWEMTest* in the UWEMTest dimension. The association with *DisabilityGroup* has an association class, *DisabilityGroupRelevance_Fcui*, with one attribute, BarrierProbability. This attribute holds the probability for that a failed outcome of the represented barrier computation means that there is an accessibility barrier for the represented disability group. In this way it is possible to use different probabilities for different disability groups, such as blind and deaf people. For example, a missing text alternative for an image introduces a barrier probability for a blind user, but not for a deaf user. Conversely, a missing text alternative for a sound resource introduces a barrier probability for the deaf user, but not for the blind user.

The association from *BarrierComputation* to *UWEMTest* represents which UWEM tests cover what a barrier computation evaluates. This association also has an association class, *UWEMCoverage*, with the attribute UWEMTestWeight. This attribute is introduced to avoid "double counting" when aggregating. That means that UWEMTestWeight holds the percentage of a test result that should be assigned to a specific UWEM test. If a barrier computation is involved in n UWEM tests, this value is currently set to $1/n$.

Time Dimension In the Time dimension there are two classes: *Minute* and *Hour*, where a minute belongs to a specific hour. Apart from the IDs, there are only the obvious attributes for holding the minute and hour values.

Date Dimension In the Date dimension there are five classes: *Date*, *Week*, *Month*, *Quarter*, and *Year*. The *Date* class is used to represent specific dates (i.e., days). There is a many-to-one association from *Date* to *Month*. Similarly there are many-to-one associations between *Month* and *Quarter* and between *Quarter* and *Year*. There are also a many-to-one associations between *Date* and *Week* and between *Week* and *Year*. *Week* is not associated with *Month* or *Quarter* since a week can cross boundaries between months or quarters. Later, domain specific knowledge may be added to the Date dimension, e.g., most current HTML version or number of days since a new WCAG version was released.

Category Dimension In the Category dimension, shown in Figure 3.4(a), there are two classes: *Category* and *Sector*. The *Category* class is used to represent different categories of web site providers. Currently, there are 13 predetermined categories

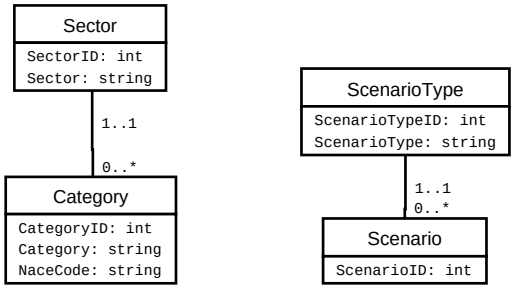


Figure 3.4: The (a) Category and (b) Scenario dimensions

such as “Banks”, “Radio stations”, and “Federal organisations”. The *Sector* class is used to represent the public and commercial sectors (these are currently the only considered sectors). A category belongs to one sector (and thus the category of public radio stations is different from the category of commercial radio stations).

Scenario Dimension In the Scenario dimension, shown in Figure 3.4(b), there are two classes: *Scenario* and *ScenarioType*. The *Scenario* class represents different simulated scenarios in the crawling. A scenario *covers* (i.e., includes) parts of, or entire, web pages that are evaluated. Apart from *Scenario*’s ID attribute, it has no attributes since only information about a scenario’s existence and associations needs to be stored in the DW. *Scenario* has a many-to-one association to *ScenarioType* which is used to represent different types of scenarios. Two different kinds of types exist: *Page scenarios* which cover entire web pages from the top to the bottom and *key use scenarios* that only cover parts of web pages. Thus, the key use scenarios are used for simulating a human user. A scenario is either a key use scenario or a page scenario.

Subject Dimension The subject Dimension, shown in Figure 3.5, is the largest and most complex. It has 16 classes: *Subject*, *PageVersion*, *TestRun*, *Server*, *OperatingSystemFamily*, *Language*, *LanguageFamily*, *Page*, *Site*, *Domain*, *SecondLevelDomain*, *TopLevelDomain*, *NutsLevel3*, *NutsLevel2*, *NutsLevel1*, and *Country*. A *subject* is a specific, tested element on a tested web page, e.g., an `image` element starting on line 3, column 4. Subjects are represented by the *Subject* class. Its attributes are Line and Col to represent the exact start position of the represented subject. Web pages are represented by the class *Page*. However, as some web pages are frequently updated, it is also necessary to consider versions of web pages. These are represented by the class *PageVersion*. A subject is considered to belong to a certain version of a web page. Thus *Subject* has a many-to-one association to *PageVersion* which itself

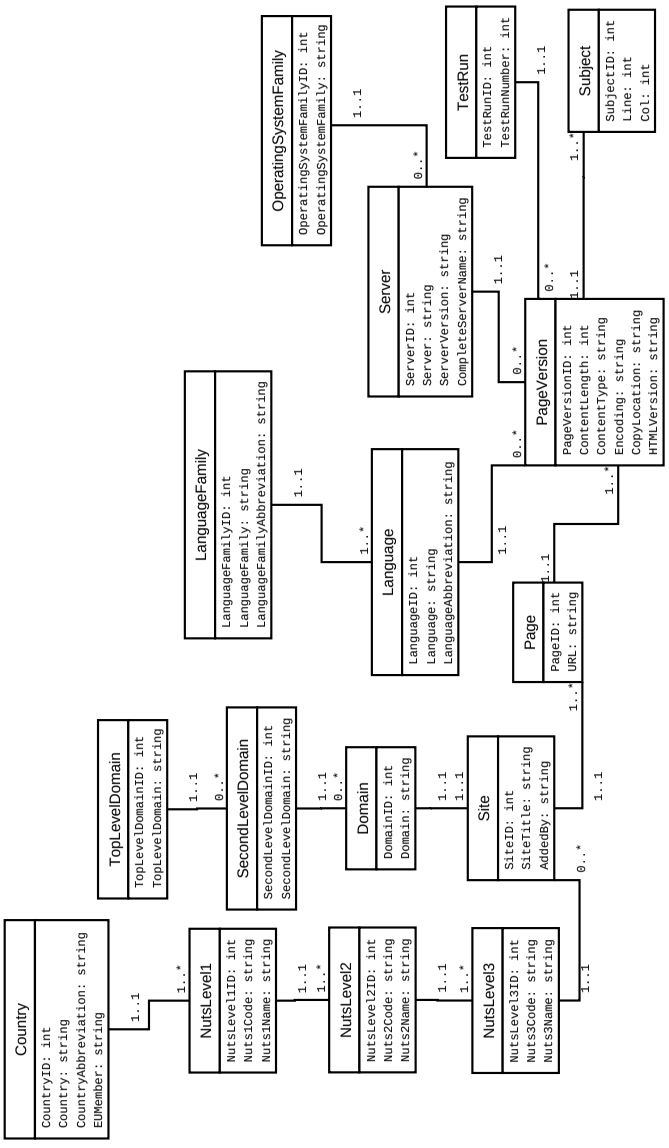


Figure 3.5: The Subject dimension

has a many-to-one association to *Page*. *Subject* also has a *many-to-many* association to *Scenario* in the Scenario dimension (not shown as each figure only shows one dimension). This is to represent which scenarios cover the subject. A subject is always covered by at least one scenario, namely the page scenario for the page version the subject belongs to. Note that this association between classes in different dimensions thus adds complexity compared to traditional DW models.

PageVersion has many-to-one associations to *Minute* and *Date* in the Time and Date dimensions. This is to represent the last modification date of the page version. *PageVersion* also has a many-to-one association to the class *TestRun* to represent from what test a represented page version was fetched from the Web. *TestRun* is used to represent the different *test runs*. A test run is the process of performing a crawl, i.e., downloading pages, and evaluating the accessibility of the pages by means of the barrier computations. When to start a new test run and when to continue using an existing is decided by the crawler; typically a new test run is started when the crawler is started with a list of web sites to consider.

PageVersion also has a many-to-one association to the class *Server*. This class is used to represent different identified web server products, such as "Apache/1.3.27 (Unix)", from the crawl. The servers are identified by considering the HTTP headers. Both the complete server string, including version information etc., and a generic server name (e.g., "Apache") are stored. *Server* has a many-to-one association to *OperatingSystemFamily* that represents different families of operating systems such as "Windows".

Finally, *PageVersion* has an association to *Language* which represents different languages such as "Swedish as spoken in Finland". To also represent the more generic languages, *Language* has a many-to-one association to *LanguageFamily* which in the previous example would represent "Swedish".

Page is associated to *Site* which represents web sites. As attributes it has a title describing the site and information about who added the site to the observatory. As previously mentioned, a web site belongs to one or more categories. Therefore *Site* has a many-to-many association to *Category* in the Category dimension. This association has an association class, *SiteCategorisation*, with the attribute CategoryWeight. This shows the percentage of a site's results that should be counted as belonging to the associated category.

Site also has an association to the class *NutsLevel3*. This class represents level 3 NUTS codes. A NUTS code [38] is a code representing a well-defined geographical area. (NUTS is short for Nomenclature des unités territoriales statistiques). The level of detail can be varied from 1 to 3. For example, Falkirk with NUTS level 3 code "UKM26" is located in Eastern Scotland with NUTS level 2 code "UKM2" in Scotland with NUTS level 1 code "UKM" in United Kingdom with country code

"UK". As motivated by the previous example, *NutsLevel3* is associated to *NutsLevel2* which is associated to *NutsLevel1* which finally is associated to *Country*.

Site also has an association to *Domain* that represents Internet domains, such as `bbc.co.uk`. *Domain* is associated to *SecondLevelDomain* that in the previous example is used to represent `co.uk` and which is associated to *TopLevelDomain* that represents top-level domains such as `uk`.

Remaining Classes As already described, there are three association classes, *DisabilityGroupRelevance_Fcui*, *UWEMCoverage*, and *SiteCategorisation*, that are not part of any dimension. Apart from these, there is only the class *TestResult* that is not in a dimension. *TestResult* is a class without attributes. It is associated to the classes *BarrierComputationVersion*, *Result*, *Subject*, *Minute*, and *Date*. Thus it is used to represent that a specific subject at a specific point in time was evaluated by a specific barrier computation version with a specific result. This, of course, means that the *TestResult* class in dimensional terms represents the *fact table*.

In the presented conceptual model, it is thus possible to represent information about web related issues such as (versions of) web pages, domains etc. It is also possible to represent information about results of the accessibility tests as well as information about the used tests. The model needs constructs with associations between classes in different dimensions and is thus more complex than traditional DW models.

3.4 Logical Model

The logical model for EIAO DW, shown in Figure 3.6, is a dimensional model based on a *star schema* [58]. There are 11 dimensions in the logical model, but two of these (the Date and Time dimensions) are duplicated, and thus if the duplicates are ignored, there are nine dimensions, as in the conceptual model.

The dimensions are formed by "merging" the classes that were described to belong together in Section 3.3. Thus, the hierarchy levels in the logical model's dimensions correspond to the classes in the conceptual model. Consider, for example, the classes *Language* and *LanguageFamily*. When pages with a specific language are considered, it is possible to *roll up* and look at language families instead (e.g., consider pages in "Swedish" instead of considering pages in "Swedish as spoken in Sweden" and pages in "Swedish as spoken in Finland" separately). The lowest level in the hierarchy for a dimension corresponds to the class that is closest to the class *TestResult* in the conceptual model.

In the logical model, primary keys have been declared. For the dimensions, the primary keys are the ID attributes at the lowest levels. Further, foreign keys have been added in the logical model for associations that cross dimensions in the conceptual

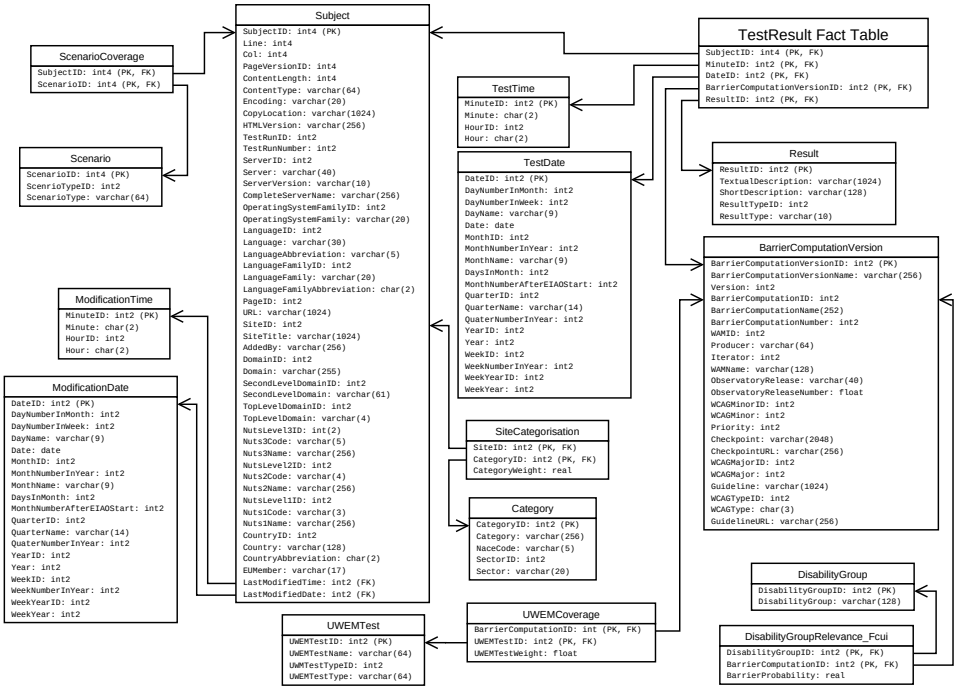


Figure 3.6: The logical model for EIAO DW

model (the associations that do not cross dimensions are represented by the internal dimension hierarchies). For the one-to-many associations, this is done by adding a foreign key to the "many side". For many-to-many associations, this is done by adding a *bridge table* [58] with two foreign keys that reference the two dimensions. If the association has an association class in the conceptual model, the attribute from this will also be added to the bridge table. Consider, for example, again the many-to-many association between *Site* and *Category* with the association class *SiteCategorisation* with the attribute CategoryWeight. The schema for the resulting bridge table becomes `CategoryWeight(SiteID, CategoryID, CategoryWeight)` where SiteID and CategoryID are foreign keys. Note that these foreign keys are not declared in the database since the ID attributes they reference are not primary keys.

Thus, there are four bridge tables in the logical model: ScenarioCoverage (that relates subjects and scenarios) and DisabilityGroupRelevance_Fcui, UWEMCoverage, and SiteCategorisation that apart from the foreign keys have attributes as the similarly named association classes in the conceptual model.

There is a single fact table, TestResult. This fact table has no measures and is thus a so-called *fact-less* fact table [58]. It is used to track events, namely results of accessibility evaluations. It has a primary key consisting of the combined for-

eign keys (referencing Result, BarrierComputationVersion, Time, Date and Subject dimensions, respectively). The dimensions that are not referenced from the fact table are *outriggers* [58] that are referenced from other dimensions or from bridge tables. This is, for example, the case for the Category dimension that is referenced from the bridge table SiteCategorisation and in that way connected to the Subject dimension.

The dimensions for representing times and dates are duplicated since they are both used as ordinary dimensions referenced from the fact table (to track when an evaluation took place) and as outriggers referenced from the Subject dimension (to track when the page holding the subject was modified). Thus there are the duplicated dimensions TestDate and ModificationDate as well as TestTime and ModificationTime. Note that in the implementation, the duplicated dimensions are just declared as views over the same base table such that the data is not physically duplicated. In fact, all dimensions and the fact table are made available through views to allow for later redefinitions.

3.5 Physical Model

In the physical model for EIAO DW, tables are declared for the different dimensions (recall that some dimensions are duplicates of others) and bridge tables in the logical model. Further, more than 30 indexes have been declared to speed up queries. Some summary tables that hold answers to the most needed queries have also been defined. These will be described in Section 3.7 where the aggregation functions for the EIAO DW are also described.

When the EIAO project is being scaled up to monitor 10,000 sites, the data sizes to handle in the EIAO DW will be very large, as previously mentioned. To be able to handle these data sets efficiently, partitioning of the data will be used. In the logical model, the by far largest tables are the fact table and the dimension tables Subject and Scenario and the bridge table ScenarioCoverage. These tables are therefore the obvious candidates for partitioning. The rest of the tables are not expected to grow so large that partitioning is necessary.

The reporting engine queries for aggregate results (see also Section 3.7) for groups of versions of web pages that were evaluated in a *specific test run*. Thus, it makes sense to use the test runs for partitioning the data. To make this easy, the TestRunID attribute can be used to decide which partition to place some given data in. This, however, requires that the primary key of the Subject dimension is changed into (SubjectID, TestRunID). The fact table should then, of course, also have a TestRunID attribute as part of its foreign key to the Subject dimension. Further, the TestRunID attribute should be added to the Scenario dimension and should be part of the primary key. Thus when a new scenario is being represented there, the TestRunID for the current test run is also inserted. Also, the ScenarioCoverage bridge table should

then reference the TestRunID attribute. When querying for results for a specific test run, it is then easy to find the partition to look into and thus avoid millions of tuples irrelevant for the query in other partitions. The use of the partitions can be handled by the stored procedures that implement the aggregation functions used by the reporting engine.

3.6 Source Data

As previously mentioned, the results of the accessibility evaluations are stored as EARL [120] which is an RDF [119] format. The EARL format is designed to describe results of tests such as these carried out by the EIAO project. Also data about the web sites is stored as RDF data. RDF is based on triples of the form (subject, predicate, object). It is possible for an object to be the subject of another triple and thus RDF can form a graph. The source data for EIAO DW is thus very different from the normal case for DWs. The ETL tool has to navigate in the graph, extract data as triples and transform it into relational data to load into the database. The triples are stored in a 3store triplestore [48]. Note that the RDF graphs easily get very large with many millions of nodes.

To load the DW, the ETL tool queries the triplestore by means of triples where one or more of the components can have the special value `None` which matches anything. The matching triples from the triplestore are then returned. Consider for example the query (x, y, z). If all of x, y, z are different from `None`, this query will return one triple (the same as the query) if it exists in the triplestore or an empty result if the triple (x, y, z) does not exist in the triplestore. If, on the other hand, z is `None`, the result of the query will be all triples in the triplestore with subject x and predicate y.

Somewhat simplified, the ETL first fetches information about test runs. For each of the test runs to load data for, the ETL then fetches information about which sites have been surveyed in the test run. For each site, it finds information on scenarios for that site and for each scenario it fetches information about tests and their outcomes. Although conceptually simple, this process is currently performing poorly when large data sets are loaded. When the triplestore holds 75 millions triples from different test runs, between 90 and 99 percent of the time is spent on waiting for the triplestore to extract triples. A faster solution is therefore needed to handle the scalability requirements for the project.

In the triplestore, both results from the tests developed by the EIAO project and from another accessibility project, called *imergo* [66], are stored. Both kinds of results are loaded into the DW.

3.7 Aggregation Functions

From the graphical user interface to the EIAO DW a number of predefined reports are available. These reports can both give an overall accessibility score and give detailed scores for tests that are relevant for a number of important UWEM tests. In both cases, the score is the result of a C-WAM [91]. The reports are either considering a single domain or groups of domains. Domains can be grouped according to country of the provider, EU membership of the country of the provider, and category of the provider. Note that the reports thus always perform some kind of aggregation. The reports will show the current score and a score for the change compared to last test run. Note that the GUI thus currently only provides these reports and does not provide access to all the data available in the EIAO DW.

For a report that considers groups of domains, the score is defined as the average of the scores for the included domains. This means that if a provider is involved in many categories, his domain will be counted in all of them. However, there is not a problem with double-counting results since we do not use already aggregated results (holding partially overlapping results) for further aggregations. Similar explanations hold for the other many-to-many relations. For a single domain, the score is calculated based on results for each of the key use scenarios as defined in [91]. Somewhat simplified, the (non-detailed) C-WAM value for a domain d, test run t (for which the key use scenarios $k_1, \ldots, k_m$ exist), and disability group g is given as $C(d,t,g) = \sum_{i=1}^{m} C'(k_i,g)/m$. Here C' of a key use scenario k (with the failed results $\{r_1, \ldots, r_n\}$) and disability group g is given by $C'(k,g) = 1 - \prod_{i=1}^{n} (1 - P_b(r_i,g))$ where $P_b(r,g)$ is a number giving the probability for that the failed test result r introduces an accessibility barrier for the disability group g. This aggregation function is very different from the aggregation functions normally used in DWs. Thus it needs special functions to be written. To make these calculations efficient and to make future changes possible without changing the reporting engine, the scores are calculated by stored procedures in the DW. The main part of the work is done by the procedures that calculate the score for a scenario and domain, respectively. The other stored procedures just use them and calculate an average. To avoid problems with precision, SQL's NUMERIC data type with up to 1000 decimal points is used in the calculation of C-WAMs for domains. The detailed C-WAMs are defined to be the ratio between failed tests for the specific UWEM tests divided by the possible number of failed tests, i.e., if two out of four tests fail, the detailed C-WAM value is 0.5. Thus significant rounding errors is not a problem for those and FLOATs can be used instead of NUMERICs.

To make the use of the stored procedures faster, *summary tables* (or *persistent caches*) have been defined for those stored procedures that calculate values for a domain. The first time a value is calculated by a stored procedure, the result is inserted

in a summary table together with the parameters to the stored procedure. Thus, the summary table for the function C defined above has columns DomainID, TestRunID, DisabilityGroupID and Result. Subsequent calls with identical parameters can then be looked up and answered immediately. To avoid that an end user faces a long response time if he is the first to calculate a given result, the stored procedures for domains are invoked for each domain during the ETL process.

3.8 Conclusion and Future Work

In this work, a data warehouse for large amounts of accessibility data has been designed and implemented. It is believed that this work is the first to develop a general and scalable business intelligence solution to the field of accessibility. By doing this, we can facilitate easy, efficient and reliable analysis of the data. To be able to do this, a simple star schema was not enough as we also had to represent many-to-many relationships. During the development, the schema was changed many times due to changes in the source data. It has thus – again – been experienced that it is difficult to develop a DW concurrently with the development of the source systems.

The solution is entirely based on open source software where the PostgreSQL DBMS has been both reliable and well-suited. However, support for materialized views is a missing feature in PostgreSQL. Built-in support for materialized views and explicitly declared bitmap indexes could have improved the performance of some slow queries.

For the implementation of aggregating accessibility score functions (C-WAMs), the stored procedure support in PostgreSQL has been valuable. This has made it possible to isolate the GUI and schema allowing for tuning of the schema. Currently, *all* C-WAM values are calculated in the DW by means of stored procedures but it seems more efficient to calculate *some* of these in the ETL.

In the current solution, RDF is used to represent the source data. When the ETL is running, most of the time is spent on extracting RDF data from the used general-purpose triplestore. Based on our (admittedly limited) experiences, RDF seems to be heavy to use for bulk-loading. The performance drops as more and more triples are present in the triplestore. In the future, it could be interesting to investigate how the RDF could be stored and extracted in a more efficient way.

The implemented solution is currently used with data for around 150 web sites. However, the EIAO project is intended to scale to 10,000 web sites being monitored on a monthly basis. The future work therefore includes to implement partitioning to handle the large data sizes (187.8 millions results each month) and possibly distribution of the DW on many machines. Also, data mining solutions should be implemented and used on the data. It will then be possible to find interesting patterns that

can improve the accessibility of web resources. These issues will be addressed in the upcoming release 2 of the entire EIAO system.

Please note that the complete EIAO system is the result of a team effort with substantial contributions from 10 organizations, see `eiao.net` [37] for details. The EIAO DW, except for the GUI, is developed by the authors. The GUI is designed and developed by a team at Agder University College, Norway.

Chapter 4

3XL: Efficient Storage for Very Large OWL Graphs

With the emergence of Semantic Web technologies like RDF and OWL, so-called triplestores that store triples of the form *(subject, predicate, object)* have become important. To store this kind of data, different kinds of data organizations have been proposed. Many of these previously proposed solutions use underlying relational databases. The data is then stored in generic schemas that can store any kind of RDF triples. This offers flexibility with respect to the data that can be stored, but the performance for data sets ("graphs") that are large suffers since most of the data is stored in a few but large tables and it becomes expensive to use those tables when searching for specific data or joining the tables. In this chapter, we describe another solution targeted towards storing data expressed in (a subset of) the OWL Lite language. The proposed triplestore called 3XL generates a specialized database schema based on information about the data to store. This is done to gain better performance for large graphs containing many millions of triples.

4.1 Introduction

It is often said that the Semantic Web has the potential to revolutionize the Web and how we use it. The content of the Semantic Web will be understandable by machines and we can thus use and access the huge amount of information on the Web in completely different ways from what we know today. To make the content available to machines, we need ways to describe meaning or semantics of data.

In the Resource Description Framework (RDF) data model used for the Semantic Web, so-called *statements* have three parts: a subject, a predicate, and an object. For example, a statement could be "Course CS123 Object-Oriented Programming is taught by John Smith". Here "Course CS123 Object-Oriented Programming" is the subject, "is taught by" is the predicate and "John Smith" is the object. To store this kind of statements, so-called *triplestores* that store triples of the form *(subject, predicate, object)* have become popular. Many such triplestores also offer further functionality such as inference where additional statements can be inferred from the stored statements. However, these triplestores may sometimes have to store many million triples and performance becomes an issue.

In this chapter, we propose the triplestore 3XL (named such to show that this triplestore can store large datasets) that scales better for the large data sets. 3XL automatically generates a specialized schema for the data based on OWL descriptions of classes and their properties. The solution uses the object-relational features of PostgreSQL [94] (in particular inheritance for tables). The use of a specialized schema instead of a general schema that stores any kind of data offers good performance, both when inserting and extracting triples. In projects where the structure of the data to store is fixed or changing rarely, the improved performance obtained with a fixed schema is a desirable and the flexibility of a general schema is not needed. And in case of changes to the structure of the data should occur, a new specialized schema can be generated with 3XL.

When generating a specialized database schema for an OWL ontology, 3XL creates a table for each OWL class. This table is then used to hold data about instances of that class. Further, 3XL may create a table for each property that does not have a maximal cardinality specified. To get good performance, 3XL inserts data into a data buffer in main memory. Only when needed due to too high memory usage or the user committing, the data is being moved into the underlying PostgreSQL database. When this happens, bulkloading techniques are used. Thus, 3XL is very well-suited for use scenarios where large amounts of triples are inserted in bulks.

The 3XL triplestore is at this stage targeted towards the basic schematic concepts such as classes (including subclasses) and properties (including domains, ranges and cardinalities). This will be enough for many projects and can be handled efficiently. However, the solution is made in a general way such that it also can be applied to different kinds of data and can be extended with other features such as inference for which only some needed type inference currently is considered.

The rest of this chapter is structured as follows. The requirements for 3XL are described in Section 4.2. In Section 4.3, an overview of 3XL's schema generation procedure is given. This is followed in Section 4.4 by more detailed descriptions of the schema generation, addition of triples, and query handling. The rows inserted into the underlying database by the proposed solution are compared to those inserted

into a generic schema similar to the one used by 3store in Section 4.5. In Section 4.6, related work is described. Section 4.7 concludes and points to future work.

4.2 Requirements

The requirements for the 3XL system are inspired by our experiences from the European Internet Accessibility (EIAO) project [37]. In the EIAO project data about the accessibility[1] of web resources is collected. Results of the accessibility evaluations are first stored as RDF [119] and EARL [120] in a triplestore and then later loaded into a multidimensional data warehouse (DW) called EIAO DW. Originally 3store [48] was used as the triplestore and data for more than one site kept together.

When around 100 web sites were considered by the EIAO project, more than 75 millions triples existed in the triplestore. Since one of the underlying MySQL tables in 3store has a row for each triple, this table is getting very big and performance suffers. When the extract–transform–load (ETL) tool was used to load the data into the EIAO DW, between 90% and 99% of the used time was spent on extracting data from the triplestore. The problem got worse as more triples were written to the triplestore.

So based on our experiences from this project, we have designed the 3XL system. 3XL is not specialized for EIAO, but instead a general tool focused on efficient storage of OWL data. More precisely, a subset of the OWL Lite constructs are supported as described below.

To generate a specialized database schema, 3XL requires information about the structure of the data to store. For these specifications, there are different options. One possibility is to use RDF Schema (RDFS) [118] which is a W3C Recommendation. However, RDFS has some drawbacks. For example, RDFS does not allow cardinalities to be specified. Further, RDFS has some rather complex aspects such as classes of classes. Another possibility is OWL [117] which is also a W3C Recommendation. OWL consists of three increasingly expressive subparts: OWL Lite, OWL DL and OWL Full. For the basic storage of data about instances, only a subset of the least expressive sublanguage, OWL Lite, is needed. OWL Lite offers some nice features such as distinction between properties that relate to other individuals (objects) and properties that relate to datatype values, and disjointness between classes, properties, individuals and data values [117]. For these reasons, OWL Lite is used to describe the data to store in 3XL. The OWL Lite constructs supported in 3XL are:

- `owl:Class`

[1] For a web page to be accessible, it must be be usable for people with disabilities. For example, a blind user using a screen reader should also be able to retrieve the information on the page so a page only containing graphic objects is not accessible. To make web resources accessible there are some guidelines to follow [122, 123].

- `rdfs:subClassOf`
- `owl:ObjectProperty`
- `owl:DataProperty`
- `rdfs:domain`
- `rdfs:range`
- `owl:Restriction`
- `owl:onProperty`
- `owl:maxCardinality`

Support for other of the left-out OWL Lite constructs can be added to 3XL later (see also Section 4.7).

The *open world* assumption from OWL is, however, not retained. In OWL, a class defined in some ontology may be extended later on in time in another ontology. For 3XL it is assumed that there exists an ontology that defines the classes once and for all at the build time for the specialized database schema. After the database schema has been built, only data about individuals (i.e., object instances) can be added and only data that fits into the schema. It is also assumed that each individual i is not both an instance of class A and class B unless one of the following conditions hold:

- A is a subclass of B.
- B is a subclass of A.
- i is also an instance of class C and C is declared to be a subclass of both A and B.

The first two conditions are needed to allow subclass relationships where it due to transitivity always holds that if a class A is a subclass of B then all instances of A are also instances of B.

The last condition says that an individual cannot be said to be an instance of both classes A and B unless a class C that is a subclass of both of these has been declared[2]. This is not a serious limitation since if i should be an instance of the classes A and B where none of them is a subclass of the other, then a class C which is a subclass of A and B just has to be created. i should then be an instance of class C which of course implies that i is also an instance of classes A and B.

The PostgreSQL DBMS [94] is an advanced open-source DBMS [109] with proven scalability and reliability. PostgreSQL also supports object-relational features

[2]Note that the `equivalentClass` construct from OWL Lite is not supported in 3XL.

(table inheritance) which are very useful for 3XL. For these reasons it is decided to let 3XL work on top of a PostgreSQL database.

In 3XL, a query is a triple of the form (s, p, o) where any of s, p, and o may have the special value denoted by $*$. The result of such a query, is a set of triples that have identical values for the three parts of the given triple. Note that $*$ matches anything in this context. Thus the query $(*, *, *)$ gives all triples in the triplestore. This kind of queries consisting of triples can be handled efficiently and it is easy to start using. For a start, this is the only way to query the triplestore, but later support for RDF query languages can be added.

4.3 Overview of 3XL

In this section, we informally describe the general idea about generating a specialized database schema for an OWL ontology. The descriptions should give an intuition about how 3XL works before this is more precisely described in the next section.

To build the database to store the data in, an OWL ontology is read. This ontology should once and for all define all classes, their parent-child relationships and their properties (including domains and ranges for all properties). In the database, a *class table* is created for each class. The class table representing the class C directly inherits from any class tables representing the parent classes of C. This means that if the class table for C's parent P has the attributes a, b, c then the class table for C also at least has the attributes a, b, c.

Two attributes are needed for each instance of any class: An ID and a URI. To have these available in all tables, all class tables, directly or indirectly, inherit from a single root class table that represents the OWL class `owl:Thing` that all other OWL classes inherit from. The class table for `owl:Thing` has the columns ID and URI. All the ID values are for convenience unique integers drawn from the same database sequence (this will be explained later).

If the class C has a `DataProperty` d with `maxCardinality` 1, the table for C has a column d with a proper datatype. For a *multiproperty* without a `maxCardinality`[3] there is a special *multiproperty table*. This multiproperty table has a column for holding the attribute values and a column that holds the IDs for the instances the property values apply to. The ID attribute acts like a foreign key, but it is not declared (this is explained below). We here denote this as a *loose foreign key*. Note that a multiproperty table does not inherit from the class table for `owl:Thing` since multiproperty tables are not intended to represent instances, but rather values for certain instances. Note also that a given multiproperty for a class C has only one multiproperty table. There will not be a multiproperty table for each subclass of C.

[3] OWL Lite only allows `maxCardinality` 0 or 1.

An `owl:ObjectProperty` is handled similarly to how an `owl:DataProperty` is handled. If the object property has `owl:maxCardinality` 1, a column for the property is created in the appropriate class table. This column holds IDs for the referenced objects. If the property is a multiproperty, the value column in the multiproperty table holds ID values.

Note that instead of using multiproperty tables, the class tables can have columns that hold arrays. In that way it is possible to represent several property values for an instance in the single row that represents the instance in question.

Example 4.3.1 *We now introduce a simple example that will be used as a running example throughout the chapter. To save space we do not use URIs as names for classes and properties introduced in the example but just short, intuitive names.*

Assume that there are three classes: Document, HTMLVersion, *and* HTMLDocument, *where* HTMLDocument *is a subclass of* Document. Document *has the properties* title *and* keyword. *The property* keyword *is the only multiproperty in this example.* HTMLVersion *has the properties* version *and* approvalDate. *Apart from the inherited properties,* HTMLDocument *has the property* usedVersion. *The property* usedVersion *is an* `owl:ObjectProperty` *with* `owl:range` HTMLVersion. *The remaining properties are all of kind* `owl:DataProperty`.

This results in the database schema drawn in Figure 4.1. Inheritance is shown with arrows as in UML. A loose foreign key is shown as a dotted arrow. Note that names of the shown classes and attributes here for convenience are as given in the example. In the actual database schema shorter, synthetic names are used. For now, please ignore the `map` *table which is explained later.*

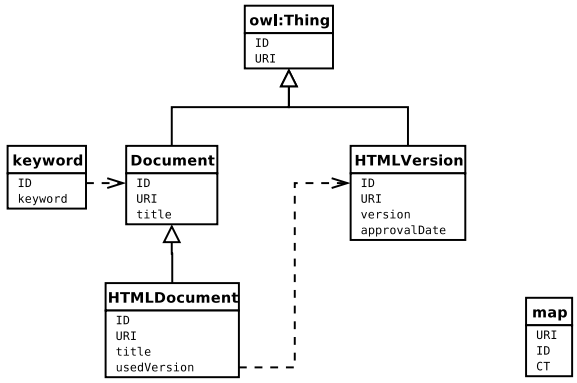


Figure 4.1: A database schema generated by 3XL

When triples are being inserted into the schema from Example 4.3.1, 3XL has to find out which class the instance in question (i.e., the subject) belongs to. This

decides which class table to insert the data into. If the property name of the triple is unique among classes, it is easy to decide. But in any case, 3XL can deduce the most specific class that the instance for sure belongs to. This most specific class might in one extreme be `owl:Thing`, but nevertheless it is then possible to represent the instance in the class table for `owl:Thing` then.

To be efficient, 3XL does not insert data from a triple into the underlying database as soon the triple is added. Instead data is held in main memory until larger amounts of data can be inserted quickly into the underlying database using bulk load mechanisms. To keep data in main memory for a while also has the advantage that the type detection described above can make a more precise guess. Note that it is a requirement in OWL Lite (and OWL DL) that there is a triple giving the `rdf:type` for each individual. Thus, a triple revealing the type should appear sooner or later and will thus often have appeared when the actual insertion into the database takes place.

In Example 4.3.1 it may, however, happen that an instance i of the class Document that has been written to the class table for Document later turns out to actually be an instance of the class HTMLDocument. But in that case it is easy to move the row representing i from the class table for Document to the class table for HTMLDocument. Here it is convenient only to have one multiproperty table for keyword since no rows have to moved from the multiproperty table. This also shows why the foreign key from multiproperty tables has to be loose. It is unknown which class table the referenced ID value is located in. However, when querying for a specific ID value for an instance of Document in Example 4.3.1, it is enough to use the SQL expression `SELECT ID FROM Document WHERE` .... PostgreSQL then automatically also looks in descendant tables. This also shows why all IDs should be unique across tables and therefore are drawn from the same sequence.

A drawback of the approach where a row representing an instance is moved from one class table T to a class table for a subclass S, is that the subclass may put a `maxCardinality 1` restriction on a property p that in the superclass is a multiproperty. In this case, the multiproperty table for p is not needed to represent data for S instances. It is then possible to let p be represented by a column in S and not by the multiproperty table that has a loose foreign key to T. However, for simplicity we keep using the multiproperty table for p if it already exists and do not add an extra column for p to the class table for S. This makes it possible to erroneously insert too many p values for an S instance, but the purpose of 3XL is to serialize the triples it is given; not to provide validation. Instead, a general OWL validator could be used in front of 3XL.

When the triplestore is queried (remember that this is done by means of a *(subject, property, object)* triple which may hold $*$ values), a similar approach can be used. If a property is given, this can reveal which class table(s) to look into. If only a subject

or object is given, it is possible to look up the URI in the `owl:Thing` class table. This is, however, potentially very expensive. For that reason, 3XL in addition to the previously mentioned class tables also has a table `map` that maps from a URI to the class table that holds the instance with that URI.

In summary, the idea compared to a table structure as the one in 3store is to have the data spread out over many tables (with potentially many columns). This makes is easier to find certain data when the table to look in can be identified easily. The tables also have a very good potential for being indexed. Indexes may be added to those attributes that are often used in queries.

4.4 The 3XL System

In this section, we first give a detailed description of how the specialized database schema of 3XL is generated. After that we describe how additions to the triplestore are handled. This is followed by a description of how queries are handled.

4.4.1 Schema Generation

In the following, we describe the handling of each of the supported OWL constructs when the specialized database schema is generated. To generate the database schema, 3XL reads an ontology and builds a model of the classes including their properties and information about subclass relationships. During the construction of this model, a mapping from property names to the most general class in the domain of the property is also built.

Based on the built model, SQL DDL statements that create the needed tables are generated and executed. Note that this SQL is not conforming to the SQL standard since it uses PostgreSQL's object-oriented extensions (see more below). The model is built first, since a property may be declared to have the domain d before the class d is described in the OWL ontology. Thus the model is only a mean to generate the final SQL, and we do therefore not describe the model in details. Instead we focus on the resulting database schema.

Note that a database schema generated by 3XL always has the table `map(uri, id, ct)`. As explained later, this table is used to make it fast to find the table that represents a given instance and the ID of the instance.

We are now ready to describe how each of the supported constructs listed in Section 4.2 are handled in the specialized database schema. Throughout the description, we assume that a database schema D is being generated for the OWL ontology O.

owl:Class An `owl:Class` in O results in a table, called a *class table*, in D. In the following text, we denote by $\mathcal{C}_X$ the class table in D for the class X in O. $\mathcal{C}_X$ is used

such that for each instance of X that is not also an instance of a subclass of X and for which data must be stored in the triplestore, there is exactly one row in X. Each represented instance has a URI and is given a unique ID by 3XL.

A special class table, $\mathcal{C}_{\texttt{owl:Thing}}$, for `owl:Thing` is always created in D. This special class table does not inherit from any other table and has two columns named ID (of type INTEGER) and URI (of type VARCHAR). Any other class table created in D will always inherit from one or more other class tables (see below) and will always – directly or indirectly – inherit from $\mathcal{C}_{\texttt{owl:Thing}}$. This implies that the columns ID and URI are available in any class table.

For other class tables than $\mathcal{C}_{\texttt{owl:Thing}}$, other columns may also be present: A class table for a class that is in the `rdfs:domain` of some property P and is a subclass of a restriction saying the `owl:maxCardinality` of the property is 1, also has a column for P. This column is only explicitly declared in the class table for the most general class that is the domain of the property. But class tables inheriting from that class table then automatically also have the column. For an example of this, refer to Example 4.3.1 where a column for title is declared in the class table for Document.

rdfs:subClassOf For classes X and Y in O where Y is a subclass of X (i.e., the triple $(Y,$ `rdfs:subClassOf`, $X)$ exists in O), there exist class tables $\mathcal{C}_X$ and $\mathcal{C}_Y$ in D as explained above. But $\mathcal{C}_Y$ is declared to inherit from $\mathcal{C}_X$ and thus has at least the same columns as $\mathcal{C}_X$. This resembles the fact that any instance of Y is also an instance of X. So when rows are read from $\mathcal{C}_X$ to find data about X instances, PostgreSQL will also read data from $\mathcal{C}_Y$ since the rows there represent data about Y instances (and thus also X instances). In Example 4.3.1 $\mathcal{C}_{\text{HTMLDocument}}$ inherits from $\mathcal{C}_{\text{Document}}$ since HTMLDocument is a subclass of Document.

Any class X defined in O that is not a subclass of another class implicitly becomes a subclass of `owl:Thing`. Thus, if no other parent is specified for X, $\mathcal{C}_X$ inherits from $\mathcal{C}_{\texttt{owl:Thing}}$ as do $\mathcal{C}_{\text{Document}}$ and $\mathcal{C}_{\text{HTMLVersion}}$ in the running example.

owl:ObjectProperty and owl:DataProperty A property (no matter if it is an `owl:ObjectProperty` or `owl:DataProperty`) results in a column in a table. If the property is an `owl:ObjectProperty`, the column is of type INTEGER such that it can hold the ID for the referenced instance. If the property on the other hand is an `owl:DataProperty`, the column is of a type that can represent the range of the property, e.g., VARCHAR or INTEGER.

If the `owl:maxCardinality` is 1, the column is placed in the class table for the most general class in the `rdfs:domain` of the property. Since there is at most one value for each instance this makes it efficient to find the data since no joining is

needed and one look-up in the relevant class table can find many property values for one instance.

If no `owl:maxCardinality` is specified, there may be an unknown number of property values to store for each instance, though. Therefore the idea about storing one property value for an instance in a column in the class table breaks. Instead, a column with an array type can be used. This is efficient if many values should be stored and all extracted at the same time. But an array is expensive to search for a specific value in. Another solution is to create a *multiproperty table*. Each row in the multiproperty table represents one value for the property for a specific instance. In a multiproperty table there are two columns: One to hold the ID of the instance that the represented property value applies to and one for the property value itself. This is the most efficient solution if it is common to search for a specific property value. This approach is illustrated for the keyword property in Example 4.3.1.

It can be left as a configuration choice if multiproperty tables or array columns should be used for properties with no explicit maximum cardinality.

rdfs:domain The `rdfs:domain` for a property is used to decide which class table to place the column for the property in in case it has a `owl:maxCardinality` of 1 or in case that array columns are used instead of multiproperty tables. In either of these cases, the column to hold the property values is placed in $\mathcal{C}_T$ where T is the domain.

If multiproperty tables are used and no `owl:maxCardinality` is given, the `rdfs:domain` is used to decide which class table holds (directly or indirectly in a descendant table) the instances for which the property values are given. So in other words, this decides where one of the IDs referenced by the multiproperty table exists. Note that no foreign key is declared in D. To understand this, recall that since there is only one multiproperty table for the given property, the most specific type of an instance that has this property may be different from the most general. So although the property has domain X, another class Y may be a subclass of X, and Y instances can then also be legally referenced by a property with range X. An example of this is seen in the running example, where keyword is defined to have the domain Document, but an HTMLDocument can also have keyword values. So in general there is not only one class table representing the range. Therefore we use a *loose foreign key*. A loose foreign key LFK_ℓ from $\mathcal{C}_X$ to $\mathcal{C}_Y$ is a column ℓ_X in $\mathcal{C}_X$ and a column ℓ_Y in $\mathcal{C}_Y$ with the constraint that if a row in $\mathcal{C}_X$ has the value v for ℓ_X, then at least one row in $\mathcal{C}_Y$ or a descendant table of $\mathcal{C}_Y$ has the value v in the column ℓ_Y. The crucial point here compared to a "normal" foreign key, is that the referenced value does not have to be in $\mathcal{C}_Y$, but can instead be in one of $\mathcal{C}_Y$'s descendants. Note that a loose foreign key is not enforced by the DBMS; this is left to 3XL to do.

If no domain is given in O for the property, it is implicitly assumed to be `owl:Thing`.

rdfs:range The `rdfs:range` is used to decide where to find referenced instances for an `owl:ObjectProperty` and to decide the data type of the column holding values for an `owl:DataProperty`. So, similarly to the case explained above, the range decides which table the other ID of a multiproperty table for an object property references by a loose foreign key. Further, when the range of a property p is known, the object of a triple where the predicate is p, can have its `rdf:type` inferred (although it in OWL Lite also has to be given explicitly).

owl:Restriction (including owl:onProperty and owl:maxCardinality) In OWL, the way to say that a class C satisfies certain conditions, is to say that C is a subclass of C' where C' is the class of all objects that satisfy the conditions [5]. The C' class can be an anonymous class. To construct an anonymous class for which conditions can be specified, the `owl:Restriction` construct is used. For an `owl:Restriction`, a number of things such as `owl:maxCardinality` can be specified.

Following the previous explanations about classes and subclasses this would lead to generating class tables for anonymous restrictions when 3XL generates the database schema. But since all instances of C' (which is actually anonymous) would also be instances of the non-anonymous class C and $\mathcal{C}_{C'}$ would thus be empty, this is more complex than needed. Instead, when 3XL generates the database schema, supported restrictions are "pulled down" to the non-anonymous subclass. So if the restriction C' of which C is a subclass, defines the `owl:maxCardinality` to be 1 for the property P by means of `owl:onProperty`, this means that P can be represented by a column in $\mathcal{C}_C$ and that no class table is generated for C'.

Currently, 3XL's restriction support is limited as only cardinality constraints are handled. As previously described, an `owl:maxCardinality` of 1 results in a column in a class table. Thus we assume that a property with `max:Cardinality` 1 only occurs once for a given subject. This is deviation from the OWL semantics where it for a property p with `owl:maxCardinality` 1 can be deduced that o_1 and o_2 are equivalent if the both the triples (s, p, o_1) and (s, p, o_2) are present.

The following table summarizes how OWL constructs from the ontology O are mapped into the database schema D.

The OWL construct ...	results in ...
owl:Class	a class table
rdfs:subClass	the class table for the subclass inherits from the class table for the superclass
owl:ObjectProperty or owl:DataProperty	a column, in a class table if the max. cardinality is 1 and in a multiproperty table otherwise
rdfs:domain	a column for the property in the class table for the domain if the max. cardinality is 1 a loose foreign key from a multiproperty table to class table otherwise.
rdfs:range	a type for the column representing the property.

4.4.2 Triple Addition

We now describe how 3XL handles triples that are inserted into a specific *model* M which is a database. M has the database schema D which has been generated as described above from the ontology O_S with schematic data. We assume that the triples to insert are taken from an ontology O_I which only contains data about instances, and not schematic data about classes etc. Note that O_I can be split up into several smaller sets such that $O_I = O_{I_1} \cup \cdots \cup O_{I_n}$ where each O_{I_i}, $i = 1, \ldots, n$, is added at a different time. In other words, unlike the schema generation which happens only once, the addition of triples can happen many times.

First, we focus on the state for M after the addition of the triples in O_I to give an intuition for the algorithms that handle this. Then, we present pseudocode in Algorithms 4.1–4.3 and explain the handling of triple additions in more details.

When a triple (s, p, o) is added to M, 3XL has to decide in which class table and/or multiproperty table to put the data from the triple. Typically, the data in a triple becomes part of a row to be inserted into M. For each different s for which a triple $(s, \texttt{rdf:type}, t)$ exists[4] in O_I and no triple $(s, \texttt{rdf:type}, t')$ where t' is more specific than t exists in O_I, a row $\mathcal{R}_s$ is inserted into $\mathcal{C}_t$.

We now consider the effects of adding a triple (s, p, o) where p is a property defined in O_S. First, assume that p is declared to have `owl:maxCardinality` 1. Then $\mathcal{R}_s$'s column for p in $\mathcal{C}_t$ gets the value $\nu(p, o)$ where $\nu(p, o)$ equals o if p is an `owl:dataProperty` and $\nu(p, o)$ equals the value of the ID attribute in $\mathcal{R}_o$ if p is an `owl:ObjectProperty`. In other words, the value of a data property is stored directly whereas an the value of an object property is not stored as a URI, but as the (more efficient) integer ID of the referenced object.

Now assume that no `owl:maxCardinality` is given for p. As previously mentioned, such properties can be handled in two ways. If array columns are used, the situation resembles that of a property with a maximal cardinality of 1. The only difference is that the column for p in $\mathcal{R}_s$ does not get its value set to $\nu(p, o)$. Instead

[4] Recall that the type must be explicitly given.

the value of $\nu(p, o)$ is added to the array in the column for p in $\mathcal{R}_s$ (but the array may also hold other values). If multiproperty tables are used, the row $\big(\iota, \nu(p, o)\big)$ where ι is the value of the ID attribute in $\mathcal{R}_s$ is added to the multiproperty table for p. In other words, the row that is inserted into the multiproperty table has a reference (by means of a loose foreign key) to the row $\mathcal{R}_s$. Further, it has a reference to the row for the referenced object if p is an `owl:ObjectProperty` and otherwise the value of the property.

So this means that for properties defined in O_S the values they take in O_I are stored explicitly in columns in class tables and multiproperty tables. For other triples, information is not stored explicitly by adding a row. If the predicate p of a triple (s, p, o) is `rdf:type`, this information is stored implicitly since this triple does not result in a row being added to M, but decides in which class table $\mathcal{R}_s$ is put.

The pseudocode listed in Algorithm 4.1 (and Algorithms 4.2 and 4.3 used by Algorithm 4.1) shows how addition of triples is handled. For a so-called *value holder* vh, we denote by $vh[x]$ the value that vh holds for x. We let the value holders hold lists for multiproperties and denote by $\circ$ the concatenation operator for a list.

Algorithm 4.1 AddTriple

Input: A triple (s, p, o)

```
 1: vh ← GetValueHolder(s)
 2: if p is defined in O_S then
 3:     if domain(p) is more specific than vh[rdf:type] then
 4:         vh[rdf:type] ← domain(p)
 5:     if maxCardinality(p) = 1 then
 6:         vh[p] ← Value(p, o)
 7:     else
 8:         vh[p] ← vh[p] ∘ Value(p, o)
 9: else if p = rdf:type and o is more specific than vh[rdf:type] then
10:     vh[rdf:type] ← o
```

When triples are being added to M, 3XL may not immediately be able to figure out which table to place the data of the triple in. For this reason, but also to exploit the speed of bulk loading, data to add is temporarily held in a *data buffer*. Data from the data buffer is then, when needed, flushed into the database in bulks. This is illustrated in Figure 4.2.

The data buffer does not hold triples. Instead it holds *value holders* (see Algorithm 4.1, line 1 and Algorithm 4.2). So for each subject s of triples that have data in the data buffer, there is a value holder associated with it. In this value holder, an associative array maps between property names and values for these properties. In other words, the associative array for s reflects the mapping $p \mapsto \nu(p, o)$. Note that if the predicate p of a triple (s, p, o) is `rdf:type`, $p \mapsto o$ is also inserted into the associative array in the value holder for s unless the associative array already maps

Algorithm 4.2 GetValueHolder

Input: A URI u for an instance

```
 1: if the data buffer holds a value holder vh for u then
 2:     return vh
 3: else
 4:     table ← ExecuteDBQuery(SELECT ct FROM map WHERE uri = u)
 5:     if table is not NULL then
 6:         /* Read values from the database */
 7:         vh ← new ValueHolder()
 8:         Read all values for u from table and assign them to vh.
 9:         Delete the row with URI u from table
10:         for all multiproperty tables mp referencing table do
11:             Read all property values in rows referencing the row for u in table and assign
                these values to vh
12:             Delete from mp the rows referencing the row with URI u in table
13:         Add vh to the data buffer
14:         return vh
15:     else
16:         /* Create a new value holder */
17:         vh ← new ValueHolder()
18:         vh[URI] ← u
19:         vh[ID] ← a unique ID
20:         return vh
```

Algorithm 4.3 Value

Input: A property p and an object o

```
1: if p is an owl:ObjectProperty then
2:     res ← ExecuteDBQuery(SELECT id FROM map WHERE uri = o)
3:     if res is NULL then
4:         res ← (GetValueHolder(o))[ID]
5:     return res
6: else
7:     /* It is an owl:DataProperty */
8:     return o
```

`rdf:type` to a more specific type than o. Actually, 3XL infers triples of the form $(s, \texttt{rdf:type}, o)$ based on predicate names, but only the most specialized type is stored (Algorithm 4.1 lines 3–4). This type information is later used to determine where to place the values held by the value holder. For a multiproperty p, the associative array will map p to a list of values (Algorithm 4.1, line 8) but for a property q with a maximal cardinality of 1, the associative array will map q to a scalar value (Algorithm 4.1, line 6). Further, 3XL assigns a unique ID to each subject which is also held by the value holder (Algorithm 4.2, line 19 when the value holder is created).

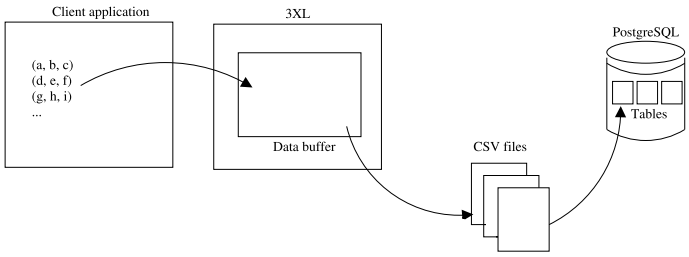


Figure 4.2: Flow of data in 3XL

Example 4.4.1 (Data buffer) *Assume the following triples are added to an empty 3XL model M for the running example:*

(http://example.org/HTML-4.0, version, "4.0")

(http://example.org/HTML-4.0, approvalDate, "1997-12-18")

(http://example.org/programming.html, title, "How to Code?")

(http://example.org/programming.html, keyword, "Java")

(http://example.org/programming.html, keyword, "programming")

Before the triples are inserted into the underlying database by 3XL, the data buffer has the following state.

http://example.org/HTML-4.0		
ID	$\mapsto$	*1*
rdf:type	$\mapsto$	*HTMLVersion*
version	$\mapsto$	*4.0*
approvalDate	$\mapsto$	*1997-12-18*

http://example.org/programming.html		
ID	$\mapsto$	*2*
rdf:type	$\mapsto$	*Document*
title	$\mapsto$	*How to Code?*
keyword	$\mapsto$	*[programming, Java]*

Here the top row of a table shows which subject, the value holder holds values for. The following rows show the associative array (which for simplicity is shown to hold the ID as well as the property values). Note that the type for http://example.org/programming.html is assumed to be Document *since this is the most general class in the domains of* title *and* keyword*.*

Now assume that the triple (http://example.org/programming.html, usedVersion, http://example.org/HTML-4.0) is added to M. Then the type detection finds that http://example.org/programming.html must be of type HTMLDocument*, so its value holder gets the following state.*

http://example.org/programming.html		
ID	$\mapsto$	*2*
rdf:type	$\mapsto$	*HTMLDocument*
title	$\mapsto$	*How to Code?*
keyword	$\mapsto$	*[programming, Java]*
usedVersion	$\mapsto$	*1*

Note how the value holder maps usedVersion *to the ID value for http://example.org/HTML-4.0, not to the URI directly. If the required* `rdf:type` *triples now are inserted, this does not change anything since the type detection has already deduced the types.*

Due to the definition of ν described above, the value holders and eventually the columns in the database hold IDs of the referenced instances for object properties. But when triples are added, the instances are referred to by URIs. So on the addition of the triple (s, p, o) where p is an object property, 3XL has to find an ID for o, i.e., $\nu(p, o)$. If o is not already represented in M, a new value holder for o is created and it is assigned a new unique ID (Algorithm 4.3, line 4). Depending on the range of p, type information about o may be inferred. If o on the other hand is already represented in M, its existing ID should of course be used. It is possible to search for the ID by using the query `SELECT id FROM` $\mathcal{C}_{\texttt{owl:Thing}}$ `WHERE uri =` o. However, for a large model with many class tables and many rows (i.e., data about many instances) this can be an expensive query. To make this faster, 3XL maintains a table `map(uri, id, ct)` where `uri` and `id` are self-descriptive and `ct` is a reference to the class table where the instance is represented. Whenever an instance is inserted into a class table $\mathcal{C}_x$, the instance's URI and ID and a reference to $\mathcal{C}_x$ are inserted into `map`[5]. By searching the data buffer which is held in memory and the `map` table (which is indexed), it is fast to look up if an instance is already represented and to get its ID if it is.

Similarly, 3XL also needs to determine if the instance s is already represented when adding a triple (s, p, o). Again the `map` table is used. If s is not already represented, a new value holder is created and added to the data buffer. If s on the other hand is represented, a value holder is created in the data buffer and given the values that can be read from the class table referenced from `map` and then $\mathcal{R}_s$ and all rows referencing it from multiproperty tables are deleted. In this way, it is easy to get the new and old data for s written to the database as data for s is just written as if it was all newly inserted (see below). This also helps, if it due to newly added data becomes evident that s has a more specialized type than known before.

When the data buffer gets full or when the user issues a commit command, the data in the data buffer is inserted into the underlying database. This is done in a bulk operation where PostgreSQL's very efficient COPY mechanism is used instead of INSERT SQL statements. So the data gets dumped from the data buffer to temporary files in comma-separated values (CSV) format and the temporary files are then read by PostgreSQL. The `rdf:type`s that are read from the value holders are used to decide which tables to insert the data into. In case, no type is known, `owl:Thing` is assumed. For unknown property values, NULL is inserted. If multiproperty tables

[5]Thus `map` corresponds to a materialized view. However, PostgreSQL does currently not support materialized views and thus 3XL has to maintain the `map` table itself.

are used, values from a multiproperty are inserted into these instead of a class table. The details of this COPY operation are not discussed here.

4.4.3 Querying for Triples

In this section, we describe how 3XL handles queries for triples in a model M. We assume that the query itself is a triple $Q = (s, p, o)$. However, any of the elements in the query triple can take the special value $*$. Since the schematic information given in O_S (for which the specialized schema was generated) is fixed, we do not consider queries for schematic information here. Instead we focus on queries for instance data inserted into M, i.e., queries for data in O_I. The result of a query consists of those triples in M where all elements *match* their corresponding elements in the query triple. The special $*$ value matches anything, but for all elements in the query triple different from $*$, all corresponding elements in a triple T in M have to be identical for T to be included in the result.

Example 4.4.2 (Queries) *Consider again the triples that were inserted in Example 4.4.1 and assume that only those (and the required triples explicitly giving the* `rdf:type`*) were inserted into* M*. The result of the query* $(*, keyword, *)$ *is the set holding the following triples:*

(http://example.org/programming.html, keyword, Java)

(http://example.org/programming.html, keyword, programming).

The result of the query $(http://example.org/HTML\text{-}4.0, *, *)$ *is the set holding the following triples:*

(http://example.org/HTML-4.0, rdf:type, owl:Thing)

(http://example.org/HTML-4.0, rdf:type, HTMLVersion)

(http://example.org/HTML-4.0, approvalDate, 1997-12-18)

(http://example.org/HTML-4.0, version, 4.0),

i.e., the set containing all the knowledge about http://example.org/HTML-4.0, including all its known types.

As there are three elements in the query triple Q and each of these can take an ordinary value or the special value $*$, there are $2^3 = 8$ generic cases to consider. We go through each of them in the following. s, p, and o are all values different from $*$. When we for a subject s say that the class table that holds s is found, it is implicitly assumed that some class table actually holds s. If this is not the case, the result is of course just the empty set. Further, we assume that all data is in the underlying database. The solution could be extended to also consider data in the data buffer, but for simplicity we here assume that the data is inserted into the database before the queries are executed.

Case (s, p, o) In this case, the query is for the query triple itself, i.e., the result set is either empty or consists exactly of the query triple. If p equals `rdf:type`, the result is found by looking in the `map` table to see if the class table holding s is $\mathcal{C}_o$ or a descendant of $\mathcal{C}_o$. This is done by using the single SQL query `SELECT ct FROM map WHERE uri =` s which can be performed fast if there is an index on `map(uri, ct)`. If s is held by $\mathcal{C}_o$ or a descendant of $\mathcal{C}_o$, Q is returned and otherwise an empty result is returned.

If p is different from `rdf:type`, the result is found by finding the ID for s (from now called $s$id) and the class table where s is inserted (by means of `map`). If that class table has a column or a multiproperty table for p, it is determined if the property p takes the value o for s. To determine this, it is necessary to look for $\nu(p, o)$ in the database as an ID is stored instead of a URI for an `owl:ObjectProperty`. If p takes the value o for s, Q is returned, otherwise the empty result is returned. So this requires an SQL query selecting the class table (if p is represented by a column) or the ID (if p is represented by a multiproperty table) from `map` and either the query `SELECT true FROM classtable WHERE id =` s`id AND` p`column =` $\nu(p, o)$ (if p is not a multiproperty), the query `SELECT true FROM classtable where id =` s`id AND` $\nu(p, o)$ `= ANY(`p`column)` (if p is a multiproperty represented by an array column), or the query `SELECT true FROM` p`table WHERE id =` s`id AND value =` $\nu(p, o)$ (if p is a multiproperty represented by a multiproperty table). In any case, only 2 SQL SELECT queries are needed and – except for the situation where p is represented by an array column – indexes on the ID and p columns can help to speed up these queries.

Example 4.4.3 (Finding a specific triple) *Assume that Q = (http://example.org/programming.html, keyword, programming) is a query given in the running example. To answer this query, 3XL executes the following SQL queries since keyword is represented by a multiproperty table.*

```
SELECT id FROM map
    WHERE uri = 'http://example.org/programming.html'
SELECT true FROM keywordTable
    WHERE id = foundID AND value = 'programming'
```

The result from the database is `true` so the triple exists in the model and 3XL returns Q itself as the result.

Case $(s, p, *)$ Also in this case, there is special handling of the situation where $p =$ `rdf:type`. Then, the `map` table is used to determine the class table $\mathcal{C}_X$ where s is located. The result set consists of all triples (s, p, C) where C is the class X or an ancestor class of X. So the only needed SQL query is `SELECT ct FROM map WHERE uri =` s. Based on the result of this and its knowledge about class inheritance, 3XL generates the triples for the result.

If p is an `owl:DataProperty`, the class table holding s is found. From this, the row representing s is found and each value for p is read. Note that if p is a multiproperty and multiproperty tables are used, the values for p are found in the multiproperty table instead by using the ID for s as a search criterion. The result set consists of all triples (s, p, V) where V is a p value for s. Again, only 2 SQL SELECTs are needed: One querying `map` and one querying for the value(s) for p from either the class table or the multiproperty table for p. Indexes on `(uri, ct)` and `(uri, id)` in `map` and on the `id`s in the classtable/multiproperty table will help to speed up these queries.

If p is an `owl:ObjectProperty`, special care has to be taken as the URIs of the referenced objects should be found, not just their IDs. The first step is to find the class table $\mathcal{C}_X$ holding s and the ID of s by means of single SELECT on the `map` table. Assume WLOG that the range of p is R. If p is represented by the column p`column` in $\mathcal{C}_X$, the query `SELECT` $\mathcal{C}_R$`.uri FROM` $\mathcal{C}_X$`,` $\mathcal{C}_R$ `WHERE` $\mathcal{C}_X$`.`p`column =` $\mathcal{C}_R$`.id AND` $\mathcal{C}_X$`.id =` s`id` is used. If p is represented by a multiproperty table `mp`, $\mathcal{C}_R$ is joined with `mp` instead of $\mathcal{C}_X$. If p is a multiproperty represented by an array column, $\mathcal{C}_X$ and $\mathcal{C}_R$ are still joined, but the condition to use is `WHERE` $\mathcal{C}_R$`.id = ANY(`$\mathcal{C}_X$`.`p`column) AND` $\mathcal{C}_X$`.id =` s`id`. The result set consists of all triples (s, p, U) where U ranges over the selected URIs.

Case $(s, *, o)$ In this case, the class table holding s is found. Then all property values (including values in multiproperty tables) are searched. The result set consists of all triples (s, P, o) where P is a property that takes the value o for s. So by iterating over the properties defined for the class that s belongs to, the previous (s, p, o) case can be used to find the triples to include. Note that also the special case $(s,$ `rdf:type`$, o)$ should also be considered for inclusion in the result set. So for this query type, an SQL query selecting the class table and the ID from `map` is needed. Further, the SQL query `SELECT true FROM mp WHERE ID =` s`id AND value =` $\nu(p, o)$ is needed for each multiproperty table `mp` representing a property p defined for s's class as is the SQL query `SELECT true FROM classtable WHERE id =` s`id AND pc =` $\nu(p, o)$ for each column `pc` representing a property p for s in the class table holding s.

Case $(s, *, *)$ In this case, the class table holding s is found by using `map`. For each property P defined in O_S (including properties represented by multiproperty tables) each of its values V for s is found. The result set then consists of all triples (s, P, V) unioned with the triples in the result set of the query $(s,$ `rdf:type`$, *)$ (found as previously described).

In this case the following SQL queries are needed: One selecting the class table and ID from `map`, the query `SELECT` p_1`column, ... ,` p_n`column FROM`

`classtable WHERE id =` s`id` if there are columns representing data properties $p_1, \ldots, p_n$ in the class table holding s, and a query `SELECT value FROM mp WHERE id =` s`id` for each multiproperty table `mp` representing a data property defined for s's class. Again, indexes on the `id` attributes in the class tables and multiproperty tables speed up the queries. Further, SQL to find the URIs for the values of object properties is needed. So for each object property q defined for the class that s belongs to and which is not represented by an array column, the following query is used: `SELECT` $\mathcal{C}_R$`.uri FROM` $\mathcal{C}_R$`,` Φ `WHERE` $\mathcal{C}_R$`.id =` Φ`.`q`column AND` Φ`.id =` s`id`. Here Φ is a multiproperty table for q or the class table holding s and R is the range of q. Indexes on the `id` attributes will again speed up the queries. If q is represented by an array column, $\mathcal{C}_R$`.id = ANY(`Φ`.`q`column)` should hold instead of $\mathcal{C}_R$`.id =` Φ`.`q`column`.

Case $(*, p, o)$ If p equals `rdf:type`, the class table $\mathcal{C}_o$ is found and all URIs are selected from it (including those in descendant tables). The result set then consists of all triples (U, p, o) where U ranges over the found URIs. This requires only 1 SQL query: `SELECT uri FROM` $\mathcal{C}_o$ which will do a scan of the entire $\mathcal{C}_o$ table (including descendants) since it has to fetch every row.

If p is different from `rdf:type`, 3XL must find the most general class G for which p is defined. If p is represented by a multiproperty table X, the tables X and $\mathcal{C}_G$ are joined and restricted to consider the rows where the column for p takes the value $\nu(p, o)$ and the URIs for these rows are selected by the query `SELECT uri FROM X,` $\mathcal{C}_G$ `WHERE X.id =` $\mathcal{C}_G$`.id AND value =` $\nu(p, o)$. If p is represented in a column in $\mathcal{C}_G$, all URIs for rows that have the value $\nu(p, o)$ in the column for p (either as an element in case p is a multiproperty represented by an array column or as the only value in case p is not a multiproperty) are selected. This is done by using either the query `SELECT uri FROM` $\mathcal{C}_G$ `WHERE` p`column =` $\nu(p, o)$ or the query `SELECT uri FROM` $\mathcal{C}_G$ `WHERE` $\nu(p, o)$ `= ANY(`p`column)`. The result set consists of all triples (U, p, o) where U ranges over the selected URIs. The first of these queries benefits from an index on the column holding data for p, but for the latter a scan is needed as we are only looking for a particular value inside an array.

Example 4.4.4 (Find subjects having a specific value for a property) *Consider the running example and assume that 3XL is given the query $Q = (*,$ keyword, programming). The most general class for which* keyword *is specified is* Document *so the SQL query* `SELECT uri FROM keywordTable,` $\mathcal{C}_{\text{Document}}$ `WHERE keywordTable.id =` $\mathcal{C}_{\text{Document}}$`.id AND value = 'programming'` *is executed. One URI is found by this query, so the triple (http://example.org/programming .html, keyword, programming) is returned by 3XL.*

Case $(*, p, *)$ If p in this case equals `rdf:type`, the result set contains all triples describing types for all subjects in the model. So for each class table $\mathcal{C}_X$, all its URIs (including those in subtables) are found with the SQL query `SELECT uri FROM` $\mathcal{C}_X$ which performs a scan of $\mathcal{C}_x$ and its descendants. The result set then consists of all triples (U, p, X) where U ranges over the URIs selected from $\mathcal{C}_X$.

If p is a data property, 3XL handles this similarly to the $(*, p, o)$ case described above with the exception that no restrictions are made for the object (i.e., the parts concerning $\nu(p, o)$ are not included in the SQL) and the values in the column representing p are also selected. But again special care has to be taken if p is an object property. It is then needed to join the class table or multiproperty table holding p values to the class table for the range R of p. Further, the column $\mathcal{C}_R$`.uri` should then be selected instead of the column representing p (this is similar to the already described $(s, p, *)$ case). For each row (U, O) in the SQL query's result, a triple (U, p, O) is included in 3XL's result set to Q.

Case $(*, *, o)$ In this case, all triples with the given o as object should be returned. This can be found by reusing some of the previously described cases. Consider that o could be the name of a class in which case type information must be returned (note that we can ignore the possibility that o is, e.g., `owl:ObjectProperty` as we have assumed that there are no queries for schematic data given in O_S). But o could also be any other kind of value that some property defined in O_S takes for some instance. In any of these cases, we can reuse the $(*, p, o)$ query handling described above. Formally, the result set is given by the following where we let $\Omega(a, b, c)$ denote the result set for the query (a, b, c) and P is the set of properties defined in O_S:

$$\Omega(*, \texttt{rdf:type}, o) \quad \bigcup \quad \left(\bigcup_{p \in P} \Omega(*, p, o) \right)$$

Case $(*, *, *)$ In this case, all triples in M should be returned. This can also be done by reusing some of the previously described cases. More concretely the result set for this query consists of a union of all type information triples and the union of all result sets for the queries $(*, p, *)$ where p is a property defined in O_S. Formally, the result set is given by the following where Ω and P are defined as above.

$$\Omega(*, \texttt{rdf:type}, *) \quad \bigcup \quad \left(\bigcup_{p \in P} \Omega(*, p, *) \right)$$

In other words, this is handled in similar way to how the $(*, *, o)$ case is handled.

4.5 Performance Discussion

In this section, we calculate the number of rows inserted by 3XL into a specialized database schema and the amount of storage used for this. The found numbers are compared to the similar numbers for a generic schema similar to the one used by the 3store system [48]. Such a generic schema can store any kind of RDF graph but does not exploit OWL specific knowledge. But since an OWL Lite graph is also a valid RDF graph, the generic schema can also store an OWL Lite graph.

We consider addition of a set of triples. The commit operation occurs as the last operation and there are no commit operations in-between the additions of triples. By A we denote all the triples given from the client application except those with `rdf:type` as their predicate. The full set, including the triples giving the types explicitly, is denoted $\bar{A}$. About A we assume that its triples describe I instances in total and that the instances *on average* have S *single properties* (a property for which an instance only can have one value, i.e., the opposite of a multiproperty). The instances on average have M multiproperties. A multiproperty on average has V different values for an instance that belongs to a class in the domain of the property. Out of all objects in the triples, we assume that L are literals. The variables are summarized in Table 4.1.

Variable	Description
A	The triples to add, excluding the required triples with the predicate `rdf:type`.
$\bar{A}$	The triples to add, including the required triples with the predicate `rdf:type`.
I	The number of unique instances in A.
S	The average number of single properties of an instance in A.
M	The average number of multiproperties of an instance in A.
V	The average number of values a multiproperty takes.
L	The number of literal values in A.

Table 4.1: Summary of variables

Using the variables from above, an average instance has $S+MV$ property values. In total there are thus $I(S + MV)$ triples in A and, assuming that $\bar{A}$ only gives the most specific type explicitly, $I(S + MV + 1)$ triples in $\bar{A}$. Addition of the triples in A thus results in the following number of rows in the underlying database when using 3XL.

Description	**Rows**	
Rows in `map`	I	
Rows in class tables	I	
Rows in multiproperty tables	$\begin{cases} 0 \\ IMV \end{cases}$	If array columns are used Otherwise

So in summary, 3XL inserts $2I$ rows if array columns are used and $I(MV+2)$ rows if multiproperty tables are used.

We now consider how many rows are inserted if $\bar{A}$ is stored in a generic schema similar to the one used in the 3store system. The generic schema uses three tables[6]: `resources(hash,uri)` which has a row for each resource, `literals(hash, literal)` which has a row for each literal, and `triples(subject, object, predicate, literal)`. So to store the triple (s,p,o), both s and p have to be represented by a row in the `resources` table, while o must be represented by a row in either the `resources` or the `literals` table. A row referencing the other tables is then inserted into `triples`. In the `hash` attributes, 8-byte hash codes are stored.

So considering the data from above, a row is inserted for each triple in $\bar{A}$, i.e., $I(S+MV+1)$ plus $I+S+M$ rows to hold the URIs of the instances and properties and L rows to represent literal values. In total this results in $I(S+MV+2)+S+M+L$ rows.

When using the current version 8.2 of the PostgreSQL DBMS, there is for each row inserted into the table t an overhead of $27+\lceil\frac{N_t}{8}\rceil$ bytes (where N_t is the number of columns in t) in the files holding data for t. Now assume that there is data about 10,000,000 instances in A and that these instances on average have 10 single properties and 5 multiproperties that on average take 5 values each. In other words, assume the following values:

$$I = 10^7 \qquad S = 10 \qquad M = 5 \qquad V = 5$$

With 3XL's specialized schema for this, the overhead from row headers is thus $\left(27+\lceil\frac{15}{8}\rceil\right)\cdot 2\cdot 10^7 = 553$MB when not using multiproperty tables and $\left(27+\lceil\frac{10}{8}\rceil\right)\cdot 2\cdot 10^7 + \left(27+\lceil\frac{2}{8}\rceil\right)\cdot 10^7\cdot 5\cdot 5 = 7.06$GB when using multiproperty tables. If the generic schema is used and there are 3 unique literal values for each instance (i.e., $L=3I=3\cdot 10^7$), the overhead is $(27+1)\cdot\left(10^7\cdot(10+5\cdot 5+2)+10+5+3\cdot 10^7\right) = 10.43$GB. This overhead is 1.5 times bigger than the overhead from 3XL's schema

[6]Compared to the generic schema used by 3store we do not have a `model` table since we only consider data for one model in one database. Further, we do not have an `inferred` attribute for the triples as 3XL does not do inference. To make the comparisons fair we do not include these items in the generic schema.

when multiproperty tables are used and it is 19 times bigger than the overhead from 3XL's schema when multiproperty tables are not used.

We now give estimates for the size of the data that is stored in the different schemas. We do not only consider the size of the raw data in $\bar{A}$. Instead we consider the size of all the data that must be stored in the schemas and thus we also include, e.g, the size of IDs for 3XL's schemas and the size of hash values for the generic schema. To give the estimates we continue to use the assumptions from above about the data. Further, we assume the following. Three of the single properties are of a numeric type. The values for these on average consist of 6 characters (i.e., digits and possibly a decimal symbol) when they are represented as a string. Two of the single properties are dates, four are strings with 15 as their average length, and one is an object reference. We assume all the multiproperties are object references. An average URI is assumed to contain 40 characters.

When we calculate the needed amount of bytes to store the data, we assume that a 4-byte length field is added in front of data with variable length (this reflects how these data types are stored in PostgreSQL). We do not consider space wasted to obtain correct alignment in the files managed by the DBMS.

First we consider 3XL when not using multiproperty tables. For each of the I instances there will be a row in `map`. This row consists of a URI which has variable length with 40 as the average and an ID which is represented as a 4-byte integer and a 4-byte reference to a class table. In total, each instance requires 4 + 40 + 4 + 4 bytes in `map`. Considering data in class tables, an average instance has three integer values (each requiring 4 bytes), two dates (each requiring 4 bytes to store), four strings of average length 15 (and thus requiring 4 + 15 bytes to store), and an object reference (stored as an integer requiring 4 bytes). Each multiproperty is an object reference and takes 5 values. Thus all the multiproperties of an instance require $5 \cdot 5 \cdot 4$ bytes. In total an average instance requires

$$\underbrace{4+40+4+4}_{\texttt{map}} + \underbrace{3 \cdot 4 + 2 \cdot 4 + 4 \cdot (4+15) + 4 + 5 \cdot 5 \cdot 4}_{\text{class tables}} = 252 \text{ bytes.}$$

With $I = 10^7$ this means that 2.35GB storage is required.

We now consider 3XL when multiproperty tables are used. The same amount of storage is needed in the `map` table. For the class tables, the calculations are very similar to the previous ones except that the bytes spent on storing multiproperty values should not be counted. Instead each multiproperty value is now stored with two integers, one to hold the ID of the instance that "owns" the value and one to store the ID of the referenced object. So in total an average instance requires

$$\underbrace{4+40+4+4}_{\texttt{map}} + \underbrace{3 \cdot 4 + 2 \cdot 4 + 4 \cdot (4+15) + 4}_{\text{class tables}} + \underbrace{5 \cdot 5 \cdot (4+4)}_{\text{multiprop. tables}} = 352 \text{ bytes.}$$

For $I = 10^7$ this means that 3.28GB storage is required.

Lastly we consider the generic schema. As before, we assume that there are three unique literals for each instance, i.e., $L = 3I$. But these have different lengths: As already said, we assume that numeric values are 6 characters long in string format, and a date must now be represented by a string of the form 'yyyy-mm-dd' which is 10 characters long. Further, all strings must have their length encoded in a 4 byte field. So we calculate the weighted average of a literal to be

$$\lambda = \frac{3}{9} \cdot (4 + 6) + \frac{2}{9} \cdot (4 + 10) + \frac{4}{9} \cdot (4 + 15) = 14.9 \text{ bytes.}$$

In the `triples` table each instance takes three 8-byte integers to hold the hash values and further there is a 1-byte boolean value to indicate if the triple holds a literal or not. In `resources` the URIs are represented (requiring 40 bytes for the string itself and 4 bytes to hold the length) together with a hash value (requiring 8 bytes). In `literals` there are three rows for each instance. Each of these rows holds a value (requiring λ bytes) and a hash value for this (requiring 8 bytes). Thus an average instance requires

$$\underbrace{8 + 8 + 8 + 1}_{\texttt{triples}} + \underbrace{4 + 40 + 8}_{\texttt{resources}} + \underbrace{3(\lambda + 8)}_{\texttt{literals}} = 145.7 \text{ bytes.}$$

For $I = 10^7$ this means that 1.36GB storage is required.

Table 4.2 summarizes the findings. It is clear that although the generic schema with its high degree of normalization is advantageous with respect to the size of the data to store, the schema compared to 3XL's without multiproperty tables has a significant overhead due to its high number of rows. Also when multiproperty tables are used, 3XL's schema requires less space than the generic but the difference is in this case much smaller.

Schema	**Size of stored data**	**Row header overhead**	**Total**
3XL without multiproperty tables	2.34GB	0.54GB	2.89GB
3XL with multiproperty tables	3.28GB	7.06GB	10.34GB
Generic	1.36GB	10.43GB	11.79GB

Table 4.2: Summary of storage requirements

The width of the rows inserted into 3XL's specialized schema and the generic schema differ. More concretely, 3XL will typically insert wider rows than if the generic schema was used since 3XL keeps single property values, and possibly also multiproperty values, in the row that represents the instance. So while the generic schema has many narrow rows in few tables (in particular a row for each triple in

one of its tables), 3XL's approach results in fewer, but wider, rows in many tables. If we assume that the I instances from above are from C classes and are evenly distributed, each of the C class tables will hold $\frac{I}{C}$ rows. If multiproperties are also evenly distributed, a multiproperty table will hold $\frac{VI}{C}$ rows. To use these tables with fewer rows is advantageous in many cases. If, for example, all triples describing a specific subject should be found, it is, as previously explained, enough to read one row from the class table and V rows from each relevant multiproperty table. Much fewer joins of large tables are needed. For data properties, no joins are needed and for object properties the tables to join have significantly fewer rows than in the generic schema. For a project like the previously mentioned EIAO project it is believed that it will be faster to use 3XL than the used 3store solution where many large joins have to take place. In the future 3XL will be implemented and its performance studied empirically.

4.6 Related Work

Many different RDF stores have been described before. In this section we describe the products that are most relevant to the current work. It should be noted that terminology may be used with different meanings in different solutions. For example, the term "class table" is not meaning the same in *RDFSuite* described below and 3XL.

An early example of an RDF store can be found in *RDFSuite* [2, 3] which is a suite of tools for validating, storing, and querying RDF data. In the querying part of the work, the language RQL is defined. In the part of the work that focuses on storing RDF data, two different representations are considered: *GenRepr* which is a generic representation that uses the same database schema for all RDF schemas and *SpecRepr* which creates a specialized database schema for each RDF schema. In [2, 3] it is concluded that the specialized representation performs better than the generic representation.

In the generic representation, two tables are used. One for resources and one for triples. In a specialized representation, RDFSuite represents the core RDFS model by means of the four tables `Class`, `Property`, `SubClass`, and `SubProperty`. Further, a specialized representation has a so-called *class table* for each class defined in the RDFS. In contrast to the class tables used by 3XL which also may store attribute values, RDFSuite's class tables only store the URIs of individuals belonging to the represented class. Both RDFSuite and 3XL use the table inheritance features of PostgreSQL for class tables. RDFSuite's specialized representation also has a so-called *property table* for each property. This is different from 3XL's approach where multiproperty tables only are used if the cardinality for the represented property is greater than 1. In RDFSuite, property tables store URIs for the source and target of each represented property value. Alexaki *et al.* [3] also suggest (but do not imple-

ment) a representation where single-valued properties with range a literal type are represented as attributes in the relevant class tables. This is similar to the approach taken by 3XL. In 3XL this is taken a step further and also done for attributes with object values.

In Broekstra *et al.*'s solution for storing RDF and RDFS, *Sesame* [18], different schemas can be used. Sesame is implemented in a way where all code that does specific data handling is isolated in a so-called *Storage and Inference Layer* (SAIL). It is then possible to provide new SAILs that can be plugged into a Sesame system. Such a SAIL can, for example, use main memory or a DBMS. Three SAILs that use a DBMS are described in [18]: 1) A generic SAIL for SQL92 compatible DBMSs, 2) a SAIL for PostgreSQL, and 3) a SAIL for MySQL.

The generic SAIL for SQL92 compatible DBMSs only uses a single table with columns for the subjects, predicates and objects. In the SAIL for PostgreSQL, the schema is inspired by the schema for RDFSuite, described above, and is dynamically generated. The RDF Schema is stored by six tables that hold information on classes, subclasses, properties, subproperties, domains, and ranges, respectively. Again, a table is created for each class to represent. Such a table has one column for the URI. A table created for a class inherits from the tables created for the parents of the class. Likewise, a table with two columns (for source and target) is created for each property. Such a table for a property inherits from the tables that represent the parents of the property if it is a subproperty. This SAIL is reported [18] to have a good query performance but disappointing insert performance when tables are created.

The SAIL for MySQL does not specialize the database schema to the RDF Schema. Instead 13 tables are always used. These tables are used to represent literals, comments, labels, properties, subproperties, resources, types, classes, subclasses, namespaces, domains, ranges, and triples, respectively. This SAIL is reported [18] to perform better than the PostgreSQL SAIL. The schema used by the PostgreSQL SAIL is closest to the schema used by 3XL. An important difference is, however, how property values are stored. In 3XL, single-valued properties are stored in the class tables and only multi-valued properties are stored in other tables.

Wilkinson *et al.* [115] suggest yet another database schema for storing RDF data. In their tool, *Jena2*, all statements can be stored in a single table with the columns `Subject`, `Predicate`, and `Object` (and a column to store a graph identifier since different graphs may be stored in the same table). In the statement table, both URIs and literal values are stored directly. Only literals and URIs that exceed a threshold need to be stored in separate tables and referenced from the statement table (to distinguish between URIs/literals and references, all stored values have a prefix that reveals the value's type). Further, Jena2 allows so-called *property tables* that store pairs of subjects and values. It is possible to cluster multiple properties that have maximum cardinality 1 together in one property table such that a given row in the

table stores many property values for a single subject. In essence these can, thus, be compared to 3XL's class tables. An important difference is, however, 3XL's use of table inheritance to reflect the class hierarchy. In Jena2, multi-valued properties may be stored in separate tables (with a column for the source and a column for the target) or in a property table as described above but with NULL values inserted. Property tables can also have columns of specific types to make the underlying storage perform better. The database schema (e.g. use of property tables) is specified at creation time and is then fixed. In 3XL, the schema is also specified once and for all, but this is done automatically based on an OWL specification of the classes. Jena2 uses the query language RDQL. Queries are transformed into sets of find operations that match patterns of the form (subject, predicate, object). This is the only kind of querying 3XL currently supports but in future work, a query language like RDQL may also be specified and implemented for 3XL.

Harris and Gibbins [48] suggest a schema with fixed tables for their RDF triple-store, 3store which is designed with a focus on performance and for that reason is unlikely to be given support for OWL [82]. One table holds all triples and has columns for subject, predicate and object. Further, this table has columns to hold the model (to be able to store different independent graphs), to represent if the object is a literal or not, and to represent if the triple is inferred or not. To normalize the schema, there are also tables for representing models, resources, and literals. Each of these three tables has two columns: One for holding an integer hash value and one for holding a text string. The triples table then reference the integer values in these three tables. This approach where all triples are stored in one table is very different from the approach taken by 3XL where the data to store is held in many different tables, namely one for each class. In fact, 3XL does not store data as triples but merges triples into n-tuples where $n \geq 2$ and regenerates the triples at query time. The purpose of this is exactly to avoid huge tables that have rows for each inserted triple. This results in good performance, but less flexibility compared to a solution like 3store which can store any kind of RDF data.

Pan and Heflin [84] suggest the tool *DLDB*. The schema for DLDB's underlying database is similar to the schema for RDFSuite. There is a table for each class with one column to represent its ID. Further, there is a table with two columns for each property. DLDB also defines views over classes. A class's view contains data from the class's table as well as data from the views of any subclasses. Instead of using such views, 3XL uses the table-inheritance properties of PostgreSQL. Note that a DLDB version for OWL also exists.

Storage of RDF data has also found its way into commercial database products. Oracle 10g manages storage of RDF in a central, fixed schema [4]. This schema has a number of tables. Most prominently, it has a table that has an entry for each part of

a triple (i.e., up to three entries are made for one triple) and a table with one entry for each triple to link between the parts in the mentioned table.

Other repositories designed for OWL (like 3XL) also exist. An example is *OWLIM* by Kirykov *et al.* [59]. OWLIM is implemented as a SAIL for Sesame. For querying and reasoning, OWLIM loads the full content of the repository into main memory. This is very different from 3XL that has its data in an underlying PostgreSQL database. A more scalable file-based version of OWLIM called *BigOWLIM* is also being developed [79]. That solution is file-based, and not based on a DBMS as 3XL is. BigOWLIM obtains some very good performance results [79]. However, we still believe in using an underlying DBMS to handle the data and thus exploit the results of decades of research and development in the database community such as atomicity, concurrency control, and abstraction.

4.7 Conclusion and Future Work

Motivated by our previous experiences with performance problems for large triplestores we have in this chapter proposed the 3XL system. 3XL is designed to provide fast storage and retrieval of OWL Lite data instances. The chapter suggests a subset of OWL Lite to support. OWL Lite was chosen since it offers some desirable properties such as cardinalities for properties and disjointness between classes and properties.

Unlike general triplestores that use a generic schema with few tables that grow very big for large graphs, 3XL creates a specialized database schema for the type of data to store. This specialized schema has a table for each OWL class of instances and a table for each property that does not have 1 as its maximal cardinality. In this way data is spread out over more tables and columns and it becomes faster to insert and extract data.

3XL is in particular making insertion of large amounts of triples fast by using bulk load technologies to insert data into the underlying database. So data is held in main memory until the user commits or memory needs to be freed. Big bulks of data are then inserted into the database. Currently, 3XL is not implemented, but it is clear that 3XL inserts fewer rows in more tables than a classical approach as 3store does. It is therefore believed that extraction of data in many use cases can be done more efficiently. Further, it is believed that 3XL's use of the data buffer and bulk loading techniques makes addition of triples fast.

There are a number of interesting directions for future work. First of all, an implementation of 3XL should be done and evaluated to collect empirical data about the performance. In the implementation focus could also be put on different details such as which indexes to create and how partitioning can be used to improve performance further.

Further, 3XL can be extended to cover a larger subset or all of the OWL Lite constructs. It seems that for a number of these constructs, support could be added by letting 3XL know them and their meaning and then "translate" queries using this information. For example, the `owl:inverseOf` construct could be handled in this way. Then if the schema ontology contains the triple $(p_1, \texttt{inverseOf}, p_2)$, 3XL would keep track of this but only store one of them, say p_1, in the underlying database. A query $Q = (s, p_2, o)$ could then automatically be rewritten to $Q' = (o, p_1, s)$ and processed as now but with a translation "back" such that each result triple (t, p_1, u) for Q' becomes (u, p_2, t) in the result set for Q. Similarly, the `owl:SymmetricProperty`, `owl:equivalentClass`, and `owl:equivalentProperty` could be supported by letting 3XL rewrite queries and/or extending result sets but without storing more data in the database. For the `rdfs:subPropertyOf` construct, a column would have to be added to the generated database schema as for any other property. But 3XL could then also consider data this column when the "parent property" was considered in queries.

Adding support for a query language would also be beneficial. Instead of traversing a path of triples to find the needed data, the end user could write a single query that finds the correct result. To find the result for such a query might require a number of joins etc. and it would be desirable to optimize the queries before executing them. A number of query languages for the semantic web already exist and it should be investigated how to support one or more of these efficiently.

Chapter 5

RELAXML: Bidirectional Transfer between Relational and XML Data

In modern enterprises, almost all data is stored in relational databases. Additionally, most enterprises increasingly collaborate with other enterprises in long-running read-write workflows, primarily through XML-based data exchange technologies such as web services. However, bidirectional XML data exchange is cumbersome and must often be hand-coded, at considerable expense. This chapter remedies the situation by proposing RELAXML, an automatic and effective approach to bidirectional XML-based exchange of relational data. RELAXML supports re-use through multiple inheritance, and handles both export of relational data to XML documents and (re-)import of XML documents with a large degree of flexibility in terms of the SQL statements and XML document structures supported. Import and export are formally defined so as to avoid semantic problems, and algorithms to implement both are given. A performance study shows that the approach has a reasonable overhead compared to hand-coded programs.

5.1 Introduction

Most enterprises store almost all data in relational databases. Additionally, most enterprises increasingly collaborate with other enterprises in long-running read-write workflows. This primarily takes place through XML-based data exchange technologies such as web services, which ensures openness and flexibility.

```
<Orders concept="B.rxc" structure="B.rxs">
   <Customer CID="1">Mini Market</Customer>
   <Order OID="1">
      <OrderLines>
         <Product PID="1" Qty="200" Date="04/03/05">Cola</Product>
         <Product PID="3" Qty="50" Date="03/01/05">Bread</Product>
      </OrderLines>
   </Order>
   <Order OID="3">
      <OrderLines>
         <Product PID="2" Qty="75" Date="05/01/05">Candy</Product>
      </OrderLines>
   </Order>
</Orders>
```

Figure 5.1: Example of an XML document

As an example, consider a database for a fictitious grocery supplier. The database has the relations Products(PID, PName), Customers(CID, CName), Orders(OID, CID), and OrderLines(OID, PID, Qty, Date) where Orders.CID references Customers. CID and OID and PID in OrderLines references OID of Orders and PID of Products, respectively. The data is as shown in the tables below.

PID	PName
1	Cola
2	Candy
3	Bread

Products

CID	CName
1	Mini Market
2	Smith's
3	Kiosk24

Customers

OID	CID
1	1
2	3
3	1

Orders

OID	PID	Qty	Date
1	1	200	04/03/05
1	3	50	03/01/05
2	2	100	04/05/05
3	2	75	05/01/05

OrderLines

Using a web-service call, a customer, e.g., Mini Market, requests an XML document with information on all their orders and the ordered products, see Figure 5.1 (for now, please ignore the concept and structure attributes in the root element). To save space, we use attributes in the shown XML, but in RELAXML the user can choose freely between elements and attributes. This document can easily be created by RELAXML. After receiving the document, the customer updates it to change the quantity of the bread ordered and the delivery date for the candy, and sends it back to

the supplier using another web-service call. The database can then be automatically updated by RELAXML to reflect the changes made to the XML document. Using traditional approaches, significant hand-coding would be necessary.

This chapter presents RELAXML, a flexible approach to bidirectional data transfer between relational databases and XML documents. Figure 5.2 shows the procedure when RELAXML exports relational data to an XML document. An export is specified using a *concept* (a view-like construct), and a *structure definition*, which specify the data to export and the structure of the exported XML document, respectively. From the concept, SQL that extracts the data, is generated, resulting in a *derived table* that can be changed by user-specified *transformations*. The resulting data is exported to an XML document with an XML Schema specified by the structure definition. Using both concepts and structure definitions separates data from structure, i.e., a single concept can be associated with multiple structure definitions. The import procedure is basically the reverse of the procedure shown in Figure 5.2 and allows for insert, update, and delete of data from the database.

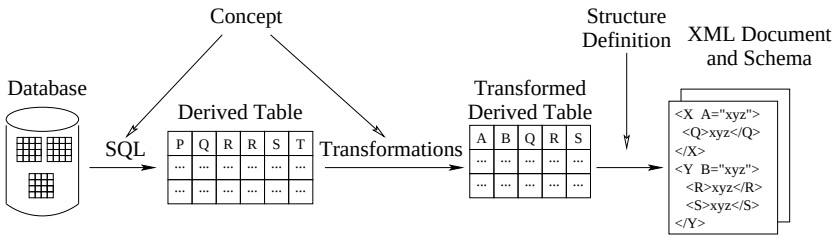


Figure 5.2: The export procedure

The SQL statement used for an export can include inner and outer joins plus filters. The structure of the XML documents is very flexible and supports, e.g., *grouping* (or *nesting*) of XML elements, data as XML elements or attributes, and additional container XML elements. Export and import are formally defined, including definitions of concepts, structure definitions, and transformations. In addition, it is specified how to determine at export time if an XML document may be imported into the database again and how an XML document must be self-contained if the data is to be imported into an empty database, so that integrity constraints are not violated. Algorithms for export and import are given. Performance studies of the DBMS independent prototype show that the algorithms are efficient, have a reasonable overhead compared to hand-coded programs, and can handle large documents ($>$ 200 MB) with a small main memory usage.

The mapping of XML data to new (specialized) relational schemas has been widely studied [12, 102]. The mapping of the result of an SQL query to an XML document (termed an *export*) has also been widely studied [20, 40, 102–104], and re-

cently SQL/XML [50] has been proposed as a standard for this mapping. However, unlike RELAXML, none of this work supports the *import* of XML documents into an *existing* database. Only few papers [13, 16, 17] have studied how to do a bidirectional (both export and import) mapping between *existing* databases and XML documents. Again, note that SQL/XML only maps from databases to XML documents. Further, some of the bidirectional approaches have limited capabilities, i.e., can only map an XML document to a single table [13]. A number of so-called *XML-enabled databases* with extensions for transferring data between XML documents and themselves exist [15, 21]. However, the solutions in these products are vendor specific and do not provide full support for transferring data into existing databases with given schemas.

There exist many *middleware products* (such as RELAXML) for transferring data between databases and XML documents [13], including products that can either export, import, or both. Examples are JDBC2XML [49], DataDesk [74] and XML-DBMS [14]. Of these, XML-DBMS is the most interesting since it can perform both import and export. It uses a mapping language to provide flexible mappings between XML elements and database columns and mappings can be automatically generated from a DTD or database schema. However, compared to RELAXML, XML-DBMS is not as scalable as it uses DOM instead of SAX, does not support inheritance or transformations, and gives no guarantee for import at export time. In [17], bidirectional transfer of data is also considered. The main differences are that [17] creates new views in the underlying database and updates through these views. Each query (tree) may need multiple new views. In contrast, we update the underlying database tables directly and do not need to modify the database schema at all. Additionally, we consider θ-joins (instead of only inner joins), we provide a performance study of an open-source prototype, and we support multiple inheritance. Compared to existing work on updating relational databases through views [30, 31], the RELAXML approach differs as 1) the SQL update statements are not known, but instead deduced from the XML document by RELAXML and 2) the needed execution order of the update statements (due to integrity constraints), is deduced from the underlying database schema by RELAXML.

The remainder of the chapter is structured as follows. Sections 5.2 and 5.3 provide definitions of basic constructs, and export and import, respectively. Sections 5.4 and 5.5 present the design of export and import, respectively. Experimental results are presented in Section 5.6. Finally, Section 5.7 concludes the chapter and points to directions of future research.

5.2 Basic Definitions

We now formally define the used constructs. When transferring relational data to an XML document, the user may want to *transform* the data in some way, e.g., by converting a price to another currency. This transformation multiplies the price by c when exporting to XML, and divides the price by c when importing from the XML.

In the following, we consider *rows* as relational tuples, i.e., a row has a number of unique attribute names (also denoted columns) and for each attribute name, an attribute value exists. For a row r and an attribute name a, $r[a]$ denotes the attribute value for a in r. Further, $\mathcal{N}(r)$ denotes the set of attribute names in r. The set of all rows is denoted $\mathcal{R}$. With this, we can define transformations formally.

Definition 5.2.1 (Transformation) *A* transformation t *is a function* $t : \mathcal{R} \rightarrow \mathcal{R}$ *that fulfills* $\mathcal{N}\big(t(r)\big) = \mathcal{N}\big(t(s)\big)$ *for all* $r, s \in dom(t)$ *where* $dom(t)$ *is the domain of* t*.*

The set of attribute names added by a transformation t is denoted $\alpha(t)$, and the set of names deleted by a transformation t is denoted $\delta(t)$. Formally, $\alpha(t) = \mathcal{N}(t(r)) \setminus \mathcal{N}(r)$ and $\delta(t) = \mathcal{N}(r) \setminus \mathcal{N}(t(r))$ for all $r \in dom(t)$. Note that for efficiency reasons, transformations are pipe-lined in the RELAXML implementation.

We now define *join tuples*, which are used for defining concepts formally. Intuitively, a join tuple defines a relation derived by joining existing relations like an SQL query, i.e., the relations to join, the join operator(s), and the join predicate(s) should be specified. For example, the join tuple for the example in Section 5.1 says that Orders and OrderLines are inner joined on the OIDs, the resulting relation is inner joined with Customers on the CIDs, and finally, this result is inner joined with Products on the PIDs.

Let θ be a theta join, and LOJ/ROJ/FOJ be a left/right/full outer join. $\Omega = I \cup O$ where $I = \{\theta\}$ and $O = \{LOJ, ROJ, FOJ\}$ is the set of RELAXML join operations (the operators in O are neither commutative nor associative).

Definition 5.2.2 (Join tuple) *A* join tuple *is a three-tuple of the form* $((r_1, \ldots, r_m), (\omega_1, \ldots, \omega_{m-1}), (p_1, \ldots, p_{m-1}))$ *for* $m \geq 1$ *and where*

1) r_i *is a relation or another join tuple for* $1 \leq i \leq m$

2) $\omega_i \in \Omega$ *for* $1 \leq i \leq m-1$

3) p_i *is a predicate for* $1 \leq i \leq m-1$.

Further, we require that if $\omega_i \in O$ *then* $\omega_j \in I$ *for* $j < i$*.*

For an $\omega \in \Omega$ and a predicate p, $A\ \omega^p\ B$ denotes the join (of type ω) where the predicate p must be fulfilled. For a given join tuple, it is then possible to compute a relation by means of the $eval$ function where

$$eval(r) = \begin{cases} eval(r_1)\ \omega_1^{p_1}\ eval(r_2)\ \omega_2^{p_2}\ \cdots\ \omega_{m-1}^{p_{m-1}}\ eval(r_m) & \text{if } r = ((r_1, \ldots, r_m), \\ & \quad (\omega_1, \ldots, \omega_{m-1}), \\ & \quad (p_1, \ldots, p_{m-1})) \\ r & \text{if } r \text{ is a relation.} \end{cases}$$

To avoid ambiguity, only one join operator from O can be used in a join tuple since they are neither commutative nor associative. If more are needed, several join tuples are used (similar to requiring parentheses in an expression).

A *concept* is used for defining which data to transfer, and thus includes a join tuple, along with a list of columns used in a projection of the relation resulting from the join tuple, a predicate to restrict the considered row set, and a list of transformations to apply. Further, as concepts support inheritance, a concept also lists its ancestors (if any). An example of concept inheritance appears in Example 5.2.1.

Definition 5.2.3 (Concept) *A* concept *is a 6-tuple (n, A, J, C, f, T) where n is the concept's caption, A is a possibly empty sequence of unique parent concepts to inherit from, J is a join tuple, C is a set of included columns from the base relations of J, f is a row filter predicate, and T is a possibly empty sequence of transformations to be applied.*

For a concept with join tuple J and parents $a_1, \ldots, a_n$, we require that the relations $D(a_1), \ldots, D(a_n)$ (defined below) are included by J.

The relation valued function D computes the base (not yet transformed) data for a concept. For a concept $k = (n, (a_1, \ldots, a_m), J, C, f, T)$, D is defined as follows, where $\nu(c)$ denotes the name of the table from which a column c originates and $cols(x)$ gives all columns in a relation x.

$$D(k) = \bigcirc_{c \in C} \rho_{[\langle k \rangle \# \nu(c) \$ c/c]}(\pi_{C \cup \{\tilde{c} \mid \tilde{c} \in cols(D(a_i)),\ i=1,\ldots,n\}}(\sigma_f(eval(J)))) \qquad (5.1)$$

First, $eval$ computes the relation that holds the data from the base relations, followed by performing a selection and then a projection of all columns included by k or any of its ancestors. Finally, a renaming schema of the columns included by k is used by means of the rename operator where # and $ represent separator characters. This 3-part naming schema (concept name, table name, column name) is necessary in order have a one-to-one mapping from the columns of $D(k)$ to the columns of the database. With the renaming schema, both table and concept names are part of the

column names of $D(k)$, which is necessary in order to separate the scopes of different concepts.

As shown above, $D(k)$ denotes a relation with the data of the concept k before transformations are applied. For a concept k with parent list $(a_1, \ldots, a_u)$ and transformation list $T = (t_1, \ldots, t_p)$, the resulting data is given by the relation valued function R defined as follows.

$$R(k) = \bigcup_{d \in D(k)} \big(\gamma(k)\big)\big(d\big), \tag{5.2}$$

where

$$\gamma(k) = \Big(\bigcirc_{n \in ((\cup_{t \in T} \alpha(t)) \setminus (\cup_{t \in T} \delta(t)))} \rho_{[\langle k \rangle \# n/n]}\big(t_p \circ \cdots \circ t_1\big)\Big) \circ \gamma(a_u) \circ \cdots \circ \gamma(a_1).$$

When a concept inherits from parent concepts, parent transformations are evaluated before child transformations. When all the transformations have been evaluated, all the attribute names they have added are prefixed with an encoding of the concept, so it is possible to distinguish between identically named attributes added by transformations from different concepts. With the definition in (5.2), a problem may emerge if a concept is inherited from twice, namely that, when transformed, an attribute included by a common ancestor could have an unexpected value, set by a transformation included by another concept. To avoid problems, we require for a concept's parent list L that $\psi(L)$ does not contain duplicates, where ψ is recursively defined as $\psi(()) = ()$ and $\psi(l_1 :: \cdots :: l_n) = l_1 :: \cdots :: l_n :: \psi\big(p(l_1)\big) :: \cdots :: \psi\big(p(l_n)\big)$ where $p(x)$ is concept x's list of parents.

Example 5.2.1 *Consider again the data in Section 5.1. We now define a concept A which extracts information on which customers have placed orders, and another concept, B, which inherits from A and adds details on the ordered products. B restricts the data to the customer with CID = 1. Thus, B extracts the data shown in Figure 5.1. We use C for Customers, O for Orders, OL for OrderLines, and P for Products.*

$$\begin{aligned} A = &(CustomersWithOrders, (), \\ &((C, O), (\theta), (C.CID = O.CID)), \\ &\{C.CID, C.CName, O.OID\}, (\mathit{true}), ()) \end{aligned}$$

$$\begin{aligned} B = &(Orders, (A), \\ &((P, OL, D(A)), (\theta, \theta), ((OL.PID = P.PID), \\ &(OL.OID = A\#O\$OID))), \{P.PID, \\ &P.PName, P.Qty, P.Date\}, A\#C\$CID = 1, ()) \end{aligned}$$

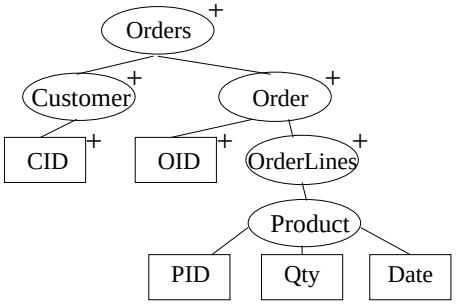


Figure 5.3: Structure definition example

Concept A has the caption CustomersWithOrders and does not inherit from other concepts. The join tuple of A states that C and O must be joined by a θ-join on the CIDs. The columns C.CID, C.CName, and O.OID are included by A. Each row from the join tuple should be included by A (each row fulfills the condition "true"). A does not use any transformations. Concept B has the caption Orders and inherits from A. The join tuple specifies how to join the relations P and OL to the relation found by A, $D(A)$. B adds three columns to those considered by A and adds a row filter such that only rows regarding a specific customer are considered.

A *structure definition* is used to define the structure (i.e., the schema) of the XML containing the data. The structure is described by means of a tree where a node represents an XML element or attribute. The structure definition for the example in Figure 5.1 is shown in Figure 5.3. A structure definition has two kinds of elements: elements that hold data but not elements, and elements that only hold other elements. A node in the structure definition can be a node that we *group by*, i.e., in the XML, elements represented by that node are coalesced into one if they have the same data values. The resulting element then holds the children of all the coalesced elements, e.g., informations on a customer and each distinct order only appear once in the XML in Figure 5.1. This is achieved by using group by nodes (marked with a +) in the structure definition in Figure 5.3. The names shown are the names used in the XML, not the relational attribute names. Below is the formal definition of structure definitions. Here, an *ordered tree with vertex set V* means that an injective order function $o : V \to \mathbb{N} \cup \{0\}$ exists.

Definition 5.2.4 (Structure definition) *A* structure definition $S = (V_d, V_s, E)$ *is an ordered rooted tree where $V_s \cap V_d = \emptyset$ and $V = V_s \cup V_d$ is the set of vertices and E is the set of edges. Members of V_s and V_d are denoted as* structural *and*

data *nodes, respectively. A vertex* $v \in V$ *is a tuple* (c, t, g) *where* c *is a name,* $t \in \{element, attribute\}$ *is the type and* $g \in \{true, false\}$ *shows if the XML data is grouped by the vertex. The root* $\rho = (c, element, true) \in V_s$ *and for every* $v = (c, t, g) \in V_s$ *it holds that* $t = element$*. For* $v = (c, t, g) \in V_d$ *it holds that if* $t = attribute$ *then* v *has no children whereas if* $t = element$ *then for each child* (d, u, h) *of* v *we have* $u = attribute$*.*

We say that a structure definition $S = (V_d, V_s, E)$ *complies* with a concept k iff for each column of $R(k)$ there exists exactly one node in V_d with identical name and the name of the root of S equals the caption of the concept k. For a concept k, a vertex $v \in V_d$ represents a column of $R(k)$ and gives rise to elements that hold *data*, while a vertex in V_s does not represent a column and gives rise to *structural* elements holding other elements. We let the function κ be a mapping between the vertices and XML tag names. Thus, the XML elements represented by v in the structure definition will be named $\kappa(v)$.

In order to represent a meaningful XML structure, a structure definition must be *valid*. For a vertex v, let $De(v)$ denote the set of descendants of v and $Ch(v)$ the set of children of v.

Definition 5.2.5 (Valid structure definition) *A structure definition* $S = (V_d, V_s, E)$ *with root* ρ *and order* o *is* valid *iff*

S1) $o(\rho) = 0$

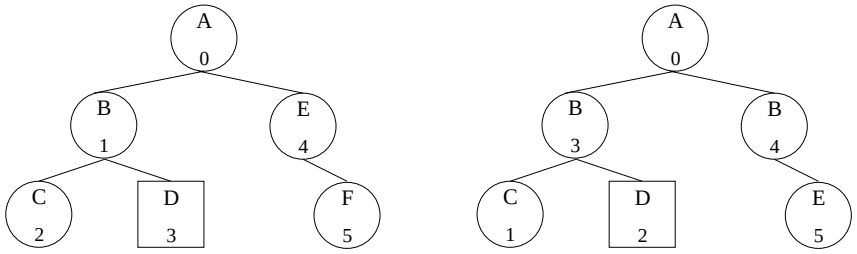
S2) For all $v \in (V_d \cup V_s)$ *we have for all* $c \in De(v)$ *that* $o(c) > o(v)$

S3) For all $a, b \in (V_d \cup V_s)$*,* $b \notin De(a)$*, we have for all* $c_a \in De(a)$ *that* $o(a) < o(b) \Rightarrow o(c_a) < o(b)$

S4) For all $v \in (V_d \cup V_s)$ *there do not exist* $c, d \in Ch(v)$ *such that* $c \neq d$ *and* $\kappa(c) = \kappa(d)$

S5) For all $(c, t, g) \in Ch(\rho)$ *we have* $t = element$*.*

Requirements S1, S2 and S3 intuitively correspond to saying that the order numbers are assigned in a depth-first fashion (this is automatically done by the RELAXML implementation and is thus of no concern for the user). Requirements S4 and S5 say that siblings should be distinguishable by having non-identical names and that the root should have only element children. Figure 5.4(a) shows an example of a valid structure definition. A node of type $element$ is represented as a circle and a node of type $attribute$ is represented as a square. A letter represents the name and a number the order. The structure definition shown in Figure 5.4(b), is not valid since the A element has two children with the name B, and the B with order 3 has children with lower order than itself.



(a) Valid (b) Not valid

Figure 5.4: Structure definitions

For a vertices v, f, p, we say that f is a *following relative* to v if f has higher order than v, and p is a *preceeding relative* to v if p has lower order than v. It is not possible to group by an arbitrary node in the tree, so we define a *valid grouping* below. Note that any valid structure definition that does not group by any nodes (the root node is trivially grouped by), is automatically a valid grouping.

Definition 5.2.6 (Valid grouping) *A* valid grouping *is a valid structure definition* $S = (V_d, V_s, E)$ *where for* $v = (n, t, g) \in (V_d \cup V_s)$ *where* $g = true$ *the following holds.*

G1) For all preceding relatives (a, b, c) *of* v*,* $c = true$*.*

G2) A following relative (a, b, c) *of* v *exists with* $c = false$*.*

G3) If a following relative that is not a descendant of v *exists, then for all descendants* (a, b, c) *of* v*, it holds that* $c = true$*.*

G4) For all children (a, b, c) *of* v *where* $b = attribute$*, it also holds that* $c = true$*.*

Requirement G1 says that when we group by a node, we have to group by its ancestors as well. Otherwise there would be no elements of the same type to coalesce in the XML. Further, the requirement ensures efficiency at import time. Without it, we risk that to regenerate a single row, many rows have to be read partly, e.g., if we in Figure 5.4(a) only grouped by E, we could have to read many B elements before the first E element, leading to a significant memory usage. Requirement G2 ensures that for each row exported, at least one element is written to the XML, ensuring that each exported row can be recreated at import time such that a grouping is not lossy. To understand requirement G3, consider Figure 5.4(a). If we group by B, we should also group by C and D. Then, an entire element, including children, represented by B can be written when the data in one row has been seen. Without G3, this would not

hold, and the writing of the element represented by E would have to be postponed. Requirement G4 ensures that a specific element's attributes only appear once in that element.

Consider again Figure 5.4(a). Now assume that we group by E. Then to have a valid grouping we must also group by A, B, C, and D, but not by F.

5.3 Export and Import

5.3.1 Export

We now define the function XML that computes XML containing the data from R. The function XML uses two auxiliary functions: $Element$, which adds an element tag, and $Content$, which adds the content of an element. These two functions depend on the structure definition used (given by the subscript). In the following, we consider the concept c with caption n and the valid grouping $\lambda = (V_d, V_s, E)$ that has the root ρ, complies with c and has order o. A string and a white space added to the XML is written in **another font** and as an underscore, respectively.

$$XML(c, \lambda) = \\ \texttt{<}n\texttt{\_concept="}\langle c\rangle\texttt{"\_structure="}\langle\lambda\rangle\texttt{">}Content_\lambda(\rho, R(c))\texttt{</}n\texttt{>} \quad (5.3)$$

The function XML adds the root element of the XML which is named after the caption of the concept c. Further, informations about the concept and structure definition are always added. The content (i.e., children) of the root element is added by $Content$. In the following, for a vertex $v = (x, y, z)$ in the structure definition, we let $v^1 = x$. Further, we let $Att(v)$ denote the ordered (possibly empty) list of attribute children of v. Then for $v = (N, t, g)$ with $Att(v) = (a_1, \ldots, a_n)$ and $Ch(v) \setminus Att(v) = \{e_1, \ldots, e_m\}$, we define $\bar{v}$ as

$$\bar{v} = \begin{cases} (v^1, a_1^1, \ldots, a_n^1) & \text{if } v \in V_d \\ (a_1^1, \ldots, a_n^1, \bar{e}_1, \ldots, \bar{e}_m) & \text{if } v \in V_s, g = true \text{ and } v \text{ has a following relative} \\ & f \notin De(v) \\ (a_1^1, \ldots, a_n^1) & \text{otherwise.} \end{cases} \quad (5.4)$$

$\bar{v}$ is used in the following to find lists of columns that should be used in projections when data to be put in the XML should be found. The function $Element_\lambda$ is defined

as

$$Element_\lambda(v,P) = \bigcirc_{\forall r \in \pi_{\bar{v}}(P)} \big(\texttt{<}\kappa(v)\texttt{\_}\kappa(a_1)\texttt{="}r[a_1]\texttt{"\_}\ldots\texttt{\_}\kappa(a_n)\texttt{="}r[a_n]\texttt{">} Content_\lambda(v, \sigma_{\bar{v}=r}(P))\texttt{</}\kappa(v)\texttt{>}\big) \quad (5.5)$$

for a relation P and a vertex v with $\bar{v} \neq ()$ and with attribute children $\{a_1, \ldots, a_n\}$ where a_i has lower order than a_j for $i < j$. If $\bar{v} = ()$, $Element_\lambda(v,P) =$ <$\kappa(v)$> $Content_\lambda(v,P)$</$\kappa(v)$>.

In (5.6), where $Ch(v) = \{e_1, \ldots, e_m\}$, $o(e_i) < o(e_j)$ for $i < j$, $(x,y,z) \in \{e_1, \ldots, e_h\}$ implies that $z = true$ and $(x,y,z) \in \{e_{h+1}, \ldots, e_m\}$ implies that $z = false$, we define the function $Content_\lambda$ for a structure node we group by.

$$\begin{aligned} Content_\lambda(v,P) = \bigcirc_{\forall w_1 : w_1 \in \pi_{\bar{e}_1}(P)} \Big(& Element_\lambda\big(e_1, \sigma_{\bar{e}_1 = w_1}(P)\big) \\ & \bigcirc_{\forall w_2 : (w_1 :: w_2) \in \pi_{\bar{e}_1, \bar{e}_2}(P)} \Big(Element_\lambda\big(e_2, \sigma_{\bar{e}_1 = w_1, \bar{e}_2 = w_2}(P)\big) \\ & \quad \ldots \\ & \quad \bigcirc_{\forall w_h : (w_1 :: \cdots :: w_h) \in \pi_{\bar{e}_1, \ldots, \bar{e}_h}(P)} \Big(Element_\lambda\big(e_h, \sigma_{\bar{e}_1 = w_1, \ldots, \bar{e}_h = w_h}(P)\big) \\ & \quad \bigcirc_{\forall r \in \sigma_{\bar{e}_1 = w_1, \ldots, \bar{e}_h = w_h}(P)} \big(Element_\lambda(e_{h+1}, \{r\}) \cdots Element_\lambda(e_m, \{r\}) \big) \Big) \cdots \Big) \Big) \end{aligned}$$
$$\text{if } v \in V_s \text{ and } g = true \quad (5.6)$$

Equation (5.6) shows that when we group by the children $e_1, \ldots, e_h$, for each distinct value of the attributes in P that are represented by e_1 and its children, we create an XML element inside which data or other elements are added recursively by means of $Element_\lambda$ which itself uses $Content_\lambda$. After each of these elements for e_1, other elements are added for those attributes that are represented by e_2 and its children. Here we have to ensure that the values for e_1 match such that we correctly group by e_1. After the elements for e_2, elements for e_3 follow and so on until elements for all group-by nodes have been added. Then elements for non-group-by nodes are added. For these nodes exactly one tuple is used for each application of $Element_\lambda$.

When using $Content_\lambda$ on non-group-by nodes, it is only given one tuple at a time. The definition of $Content_\lambda$ is

$$\begin{aligned} & Content_\lambda(v, \{r\}) = \\ & \quad Element_\lambda(e_1, \{r\}) \cdots Element_\lambda(e_m, \{r\}) \quad \text{if } v \in V_s \text{ and } g = false, \end{aligned} \quad (5.7)$$

where $Ch(v) \setminus Att(v) = \{e_1, \ldots, e_m\}$ and for $i < j : o(e_i) < o(e_j)$. That is, when not grouping by $v \in V_s$, we simply add one element for each element child of v.

Now we define $Content_\lambda$ for nodes in V_d. But from (5.5) we have that whenever $Content_\lambda$ is given a node $v \in V_d$, the given data has exactly one value for the attribute that v represents. Thus, all that $Content_\lambda$ should do is to add this value: Therefore $Content_\lambda(v, P) = Content_\lambda\big(v, \pi_v(P)\big)$ if $|P| > 1$ and $v \in V_d$ and $Content_\lambda\big(v, \{r\}\big) = r[v]$ if $v \in V_d$.

For an example, consider again the data in Section 5.1 and the structure definition in Figure 5.3 where the order of nodes is increasing from top to bottom, left to right.

5.3.2 Import

In the following, we refer to different states of the database. The value of the function D from (5.1) depends on the state of the database and we therefore refer to the value of $D(c)$ in the specific state s as $D_s(c)$. Now consider an XML document

$$\begin{aligned} X \;=\; & \texttt{<}n\texttt{\_concept="}\langle c\rangle\texttt{"\_structure="}\langle s\rangle\texttt{">} \\ & \cdots\texttt{</}n\texttt{>}, \end{aligned} \tag{5.8}$$

created by means of the concept c. By $D^{XML}(X)$ we denote a table with column names as $D(c)$ that holds exactly the values resulting when the inverse transformations from c have been applied to the data in X. It is a requirement for importing Xthat the transformations of c are invertible. This is, in the general case, undecidable and, thus, it is left to the user to ensure this. In the following, we do not consider the possible impacts of triggers and assume that foreign keys can only reference primary keys.

We now give definitions of inserting and updating from the XML. The definitions give the states of the database before and after the modifications, not the individual operations performed on the database. When inserting, the data from the XML file should be inserted into tables in the database, e.g., it should be possible to insert the data in the XML in Figure 5.1 into a database with a schema similar to that described in Section 5.1.

Definition 5.3.1 (Inserting from XML) *For a given database,* inserting *from the XML document X in* (5.8) *is to bring the database that holds the relations used by c from a valid state a to a valid state b where $D_b(c) = D_a(c) \cup D^{XML}(X)$ such that the only difference between a and b is that tuples may have been added to relations used by c.*

The data in D^{XML} or some of it can be in the database before the insertion but only in such a way that no updates are necessary, i.e., data is only inserted. We now define

updating from the XML. If an exported XML document is changed and the changes should be propagated to the database, updating is used. For example, the quantity Cola in line 5 in Figure 5.1 can be changed to 300. In that case, updating results in the database with the value 300 for Qty in the corresponding row (where OID = 1 and PID = 1) in the OrderLines table shown in Section 5.1.

Definition 5.3.2 (Updating from XML) *Consider the XML document X in* (5.8) *and assume that k is the set of renamed primary keys in the relations used by the concept c.*

For a given database that holds the relations used by c and tuples such that $\pi_k(D^{XML}(X)) \subseteq \pi_k(D_a(c))$, updating *from the XML document X is then, by only updating tuples in base relations used by c, to bring the database from a valid state a to a valid state b where for any tuple t*

$$t \in D^{XML}(X) \Rightarrow t \in D_b(c),$$

$$\big(t \in D_a(c) \wedge \pi_k(\{t\}) \nsubseteq \pi_k\big(D^{XML}(X)\big)\big) \Rightarrow t \in D_b(c)$$

$$t \notin D^{XML}(X) \wedge t \notin D_a(c) \Rightarrow t \notin D_b(c).$$

Informally, the first requirement says that a tuple read from the XML will be in the database after the updating. The second says that a tuple which is in the database before the updating, but not in the XML, is left untouched in the database. The third says that new tuples, that are neither in the database or XML, are not introduced in the database. It is also possible to combine inserting and updating, such that tuples are updated if possible and otherwise inserted. This is called *merging*.

Definition 5.3.3 (Merging from XML) *Consider the XML document X in* (5.8) *and assume that k is the set of renamed primary keys in the relations used by the concept c.*

For a given database that holds the relations used by c, merging *from the XML document X is then, by only adding tuples to or updating tuples in base relations used by c, to bring the database from a valid state a to a valid state b where for any tuple t*

$$t \in D^{XML}(X) \Rightarrow t \in D_b(c),$$

$$\big(t \in D_a(c) \wedge \pi_k(\{t\}) \nsubseteq \pi_k\big(D^{XML}(X)\big)\big) \Rightarrow t \in D_b(c)$$

$$t \notin D^{XML}(X) \wedge t \notin D_a(c) \Rightarrow t \notin D_b(c).$$

Notice that the requirement $\pi_k(D^{XML}(X)) \subseteq \pi_k(D_a(c))$ from Definition 5.3.2 is not present in Definition 5.3.3. In Definition 5.3.3 it is implied by $t \in D^{XML}(X) \Rightarrow t \in D_b(c)$ that a tuple in the database in state a, for which a tuple t with matching values for the primary keys exists in $D^{XML}(X)$, is replaced in the state b by t.

Further, *deletion* via XML is supported under some circumstances. To delete, we use a *delete document* which has the same structure as XML documents generated by RELAXML. As many as possible of the tuples in the database with data present in the delete document will be deleted. The reason that everything is not always removed, is that foreign key constraints may inhibit this.

Since delete documents must have the same structure as the XML documents being exported/imported by RELAXML, D^{XML} can be computed for identification of the data to delete from the base relations.

Definition 5.3.4 (Deleting via XML) *For a given database* deleting *base data by means of the XML document X in* (5.8)*, is to bring the database that holds the relations used by the concept c from a valid state a to a valid state b. This should be done by deleting the tuples contributing to $D^{XML}(c)$ from the base relations used by c but without violating the integrity constraints of the database.*

It should hold that $t \in D^{XML}(c) \Rightarrow t \notin D_b(c)$ unless some value in t is referenced by a foreign key not included by c and in a relation that has not been declared to set the foreign keys to a null or default value or delete referencing tuples if t is deleted.

The deletion of tuples from relations used by c may lead to updates or deletion of tuples of other relations in the database according to the integrity constraints defined on the database. Apart from this, only tuples in relations used by c will be deleted.

5.4 Design of Export

We now focus on the design and implementation of RELAXML. When exporting, an SQL statement for retrieval of the data is created based on the concept. Figure 5.5 shows the RELAXML flow when exporting. A JDBC [96] `ResultSet` is decorated with an iterator and a number of transformations. If the XML should be grouped by one or more elements, a database sort is required, since we do not want to hold all data in main memory when writing. Finally, the data rows are handed to an XML writer.

5.4.1 SQL Statements

The SQL statement to extract data from the database is generated from the concept of the export. SQL statements for parent concepts appear as nested SQL statements in the FROM clause. Note that due to inheritance the actual columns and row filter of the concept consist of the columns and row filters of parent concepts together with included columns and row filter defined in the concept itself.

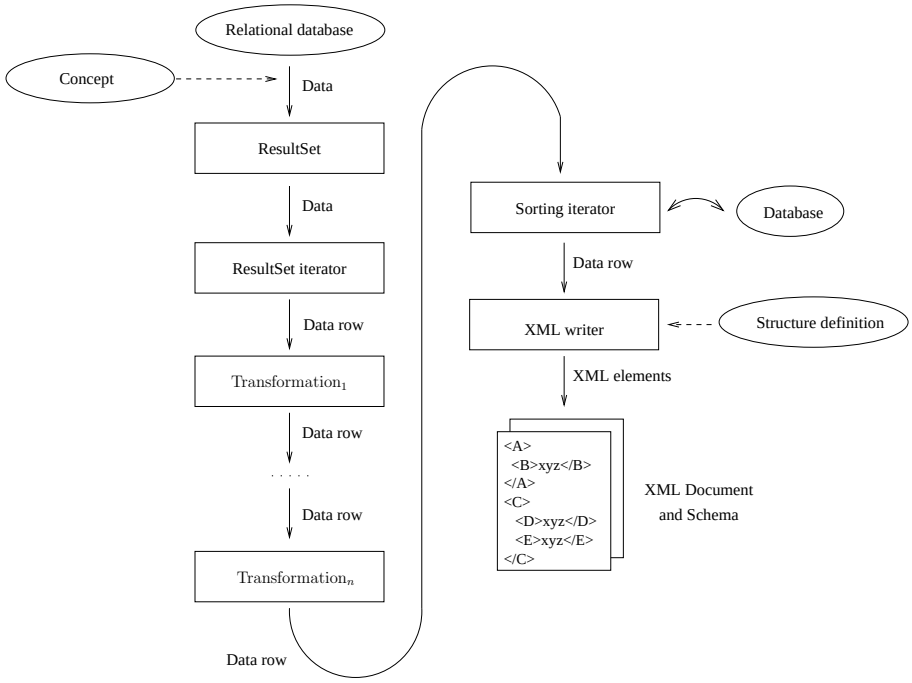


Figure 5.5: The flow of data in an export

Example 5.4.1 *Canonically, the SQL for the retrieval of the data of concepts A and B from Example 5.2.1 is as follows. Note how the three-part naming schema is imposed and how the SQL code of parent concepts appears as nested sub-queries. Modern DBMSs will, when optimizing, flatten this expression out to a regular four-way join.*

```
-- Concept A --
SELECT C.CID AS A#C$CID, C.CName AS A#C$CName,
O.OID AS A#O$OID FROM C JOIN O ON (C.CID = O.CID)

-- Concept B --
SELECT A#C$CID, A#C$CName, A#O$OID, P.PID AS B#P$PID,
P.PName AS B#P$PName, OL.Qty AS B#OL$Qty,
OL.Date AS B#OL$Date FROM
OL JOIN P ON (OL.PID = P.PID) JOIN
(SELECT C.CID AS A#C$CID, C.CName AS A#C$CName,
O.OID AS A#O$OID FROM C JOIN O ON (C.CID = O.CID)) RXTMP0
ON (OL.OID = A#O$OID) WHERE (A#C$CID = 1)
```

The code generation shown above generalizes to situations with multiple inheritance. In the implementation, the generated SQL does not contain the long names with #'s

X	Y	Z
1	A	null
2	B	1
3	C	1

Figure 5.6: Data where dead links can arise

and $'s. Instead COL0, COL1, ... are used to avoid problems with DBMSs that do not support special characters and long names. RELAXML automatically handles this mapping.

5.4.2 Dead Links

When exporting a part of the database, we may risk that the data is not self-contained. If an element represents a foreign key it may reference data not included in the XML document. We refer to such a situation as the referencing element having a *dead link*. Figure 5.6 shows an example where dead links can arise. In the example, Z is a foreign key referencing X. The data in the figure has no dead links but if the tuple with $X = 1$ is removed (this happens if the used concept only considers rows with $X \geq 2$), the data set contains two dead links since $X = 1$ is referenced by the other tuples.

A dead link does not limit the possibility of updates during import assuming that the element referenced in the dead link still exists in the database. Insertion into a new database is limited by a dead link because of integrity constraints.

In order to *detect dead links* we use Algorithm 5.1. Here, we iterate through each table used in the derived table. We find the foreign keys and the corresponding referenced keys. In line 5 we find the dead links of the derived table.

When *resolving dead links*, the goal is to expand the selection criteria such that the missing tuples are added. This may be done by adding OR clauses. Note that the SQL statement consists of possibly many nested SELECT statements in the FROM clause and that because of the scope rules, specialized concepts may include a WHERE clause on the columns of ancestor concepts. For this reason, an expansion of the condition must in some cases be added several places in the SQL. This means, that instead of the SQL statement described in Section 5.4.1, we move the WHERE clauses of the nested queries to the outermost query where they are AND'ed together. The dead link resolution algorithm shown in Algorithm 5.2 invokes Algorithm 5.1 to find dead links, manipulating the WHERE clause such that the referenced tuples are included. When a fix point is reached, all the dead links are resolved.

Algorithm 5.1 Detect dead links

1: **for** each table T part of the derived table DT **do**
2: find the sequence $A = (a_1, \ldots, a_n)$ of foreign keys in T also included in DT
3: find the corresponding sequence $B = ((b_{1,1}, \ldots, b_{1,m_1}), \ldots, (b_{n,1}, \ldots, b_{n,m_n}))$ of candidate keys that are referenced by the foreign keys in A where B is also in DT
4: **for** each $a_i \in A$ **do**
5: $M \leftarrow$ SELECT DISTINCT a_i FROM DT AS DT1 WHERE NOT EXISTS (SELECT $b_{i,1}, \ldots, b_{i,m_i}$ FROM DT AS DT2 WHERE DT1.a_i = DT2.$b_{i,1}$ OR ... OR DT1.a_i = DT2.b_{i,m_i}) AND a_i IS NOT NULL
6: $result[T][a_i] \leftarrow M$
7: **return** $result$

Algorithm 5.2 Resolve dead links

1: determine the derived table DT which may have dead links
2: $deadlinks =$ find dead links in DT by means of Algorithm 5.1
3: **for** each $deadlinks[t]$ **do**
4: *// Consider tables contributing to DT*
5: **for** each $deadlinks[t][a]$ **do**
6: *// Consider columns*
7: **for** each value v in $deadlinks[t][a]$ **do**
8: *// Consider rows (i.e., cells)*
9: expand DT's SQL expression with "OR $a = v$"
10: **if** DT's SQL has been expanded **then**
11: Invoke recursive call and find new DT to resolve dead links in
12: **else**
13: **return** DT

5.4.3 XML Writing

A desirable characteristic is not to rely on having all data stored in memory at one time. Thus, the algorithm for writing the XML works such that whenever it gets a new data row, it writes out some of the data to the XML. If grouping is not used, all the data represented in a data row is written to the XML when a data row is received. If grouping is used, some of the data might already be present in the current context in the XML and should not be repeated. To ensure this, the write algorithm compares the new row to write out and the previous row that was written. When grouping is used, it is a precondition that the data rows are sorted by the columns corresponding to the nodes that we group by. This is ensured by a DBMS-based sorting iterator. When

grouping by more than one node, the sort order is determined by the order of the structure definition. The procedure for writing the XML is outlined in Algorithm 5.3.

Algorithm 5.3 Write the XML

- Write the root element including information about concept and structure definition.
- For each data row do:
  - Find a node we do not group by or a mismatching node (considering this and the previous row). The node should have the lowest order possible. If no rows have been seen before, we let this be the node with the lowest order apart from the root. Denote this node x.
  - If we at this point have any unmatched opening tags for x and/or nodes with higher order than x, print closing tags for them.
  - Print opening tags for ancestor nodes of x that are not already open.
  - For x and each of its siblings of type element and container and with higher order do:
    - Print a **<** followed by the tag name for the node
    - Print each tag name for the node's attribute children followed by **="**, the data for the attribute node and a **"**.
    - Print a **>**.
    - If the node is an element, print its data. Else if the node is a container, perform the inner most steps recursively for all its element and container children.
    - If the node is an element or a container that we do not group by or that has a sibling with higher order, print a closing tag for the node.
- Print closing tags for any unmatched opening tags (this at least includes the root tag).

To support type checking and validation on the XML document structure, RELA-XML can generate an XML Schema based on the concept and structure definition. The user chooses at export time if a Schema should be generated or if he wants to use an existing Schema.

In order to generate the XML Schema for an export, we need information on the available columns, their types and the structure of the XML document. A `Concept` object reveals the columns and their SQL types (the types are from `java.sql.Types`) when the `getDataRowTemplate()` method is invoked, and the struc-

ture of the XML document is given in the structure definition. For each column in the data row template, a data type is generated in the XML Schema. The generated type is a `simpleType` which is restricted to the XML Schema type that the columns SQL type is mapped to. It is, however, necessary to take special considerations if the column can hold the value null, i.e., if the column is *nullable*. When exporting, RELAXML will write the null value as a string chosen by the user. But if, for example, a column of type integer is nullable, then the type generated in the XML Schema should allow both integers and the string used to represent the null value. Therefore, the generated type should be a `union` between integers and strings restricted to one string (the one chosen by the user).

The `StructureDefinition` holds a tree of structure nodes representing the tree structure of the XML document. The Schema is generated by traversing this tree. Three types of nodes exist: container nodes, element nodes and attribute nodes. The container nodes have no associated data type since their only content is elements. Elements and attributes on the other hand have associated data types since they have text-only content. These associated data types are those generated as described above.

When container nodes are treated, the Schema construct `sequence` is used. For a container that we do not group by, all its children (which by definition also are not grouped by) are declared inside one `sequence`. This ensures that in the XML instances of the considered element type each has exactly one instance of each of its children element types.

For a container that we do group by there are more considerations to take. If we consider a node x which we group by and which has at least one descendant which we do not group by, then, for each child we group by, we start a new nested `sequence` with `maxOccurs='unbounded'`. These `sequence`s are not ended until all children of x have been dealt with. All children of x that we do not group by are declared inside one `sequence` which has the attribute `maxOccurs='unbounded'`. For a structure definition as the one shown in Figure 5.7 where we assume that we group by A, B and C, these rules ensure that in the XML an instance of B is always followed by one instance of C which is followed by one or more instances of D. It is, however, possible for an instance of C to follow an instance of D as long as the C instance is followed by at least one other instance of D.

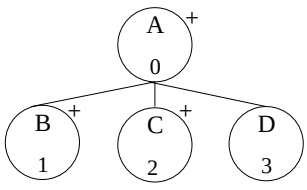


Figure 5.7: Example of a structure definition

If we consider a container x where we group by x and all its descendants, then all element types for children of x are declared inside one single `sequence`.

5.5 Design of Import

The flow of the import operation is the reverse of the flow in Figure 5.5, except that no sorting iterator is needed. Thus, the XML data is converted to data rows as the XML document is read. These data rows are sent through the inverse transformations and finally an *importer* takes appropriate action based on the data rows. We now discuss insertion, update, and deletion via XML documents.

The user may specify a commit interval such that the importer commits for every n data rows. If $n = \infty$ we may take advantage of deferrable constraints and may do a complete roll-back in case of problems such as violated integrity constraints. If $n \neq \infty$ we cannot defer the deferrable constraints and cannot do a complete roll-back.

We extend the description of concepts given in Section 5.2 by allowing a column to be marked "not updateable". If this is the case, the data in the database for that column will not be modified by RELAXML.

5.5.1 Requirements for Importing

For a concept to be *insertable*, *updateable*, or *deleteable*, it must fulfill the following requirements.

The **common requirements for insert, update, and delete** are:

c1) all transformations have an inverse;

c2) all columns used in joins occur in the derived table.

Requirement c1) is obvious. Requirement c2) is needed to support θ-joins. If we do not have values for all join columns, we cannot insert/update rows in the underlying tables. If only equijoins were supported, values for half the join columns could be derived.

The **requirements for insert** are:

i1) all non-nullable columns without default values from included tables are in the export;

i2) if a foreign key column is included, then the referenced column is also included;

i3) the exported data contains no dead links;

i4) if all deferrable and nullable foreign keys are ignored, there are no cycles in the part of the database schema used in the export.

Requirement i1) corresponds to Date's rule for insert on a view with projection [30]. Requirements i2) and i3) ensure that inserts do not cause foreign key constraint violations due to foreign keys pointing to non-existing rows. Requirement i4) ensures that it is possible to insert rows into the underlying tables in an order that avoids immediate foreign key constraint violations.

The **requirements for update** are:

u1) each included table has a primary key which is fully included in the export;

u2) primary key values are not updated.

Requirement u1) is a restriction on Date's rule for updating a view with projection [30]. Requirement u2) ensures that primary keys can be used to identify the tuples to update. To ensure that primary keys are not updated, a checksum transformation may be used to include a primary key checksum in the XML file. The **requirements for delete** are the same as for update.

If a concept A uses inheritance, all A's ancestors must be insertable or updateable for A to be insertable or updateable, as we want to ensure that the requirements described above are fulfilled for each row in the export. Otherwise, we would risk that for a concept c, one parent p_1 included some, but not all, columns from a table t required for c to be importable, while another parent p_2 included the remaining columns from t required for c to be importable. But if p_1 only includes the rows where the predicate b is fulfilled whereas p_2 includes those rows where b is not fulfilled, we cannot combine the resulting row parts to insertable rows.

In summary, concepts are much more flexible than modification through SQL views [30], e.g., multiple tables may be updated and consistency is guaranteed. Compared to Date's general specification of modification through views [30] we have stricter requirements on projection for insert and update and do not consider SQL statements with union, intersect, and difference. Concepts involving only joins of tables are insertable and updateable in the same way as views in Date's general specification. Compared to Date we support inheritance and guarantee that updates are consistent as discussed next.

5.5.2 Avoiding Inconsistency

Since the XML document may hold redundant data originating from the same cell in the database, it is a risk that the user makes an *inconsistent update*, e.g., if the same column from a table is selected twice. When the user is editing the XML, he is indirectly making updates to the transformed derived table. But since the derived table can contain redundant data, it is only in 1NF in the general case.

To detect inconsistent updates, we capture which values in the database are read from the XML, as further updates on these would be inconsistent. Thus, for all updated or accepted values (those that were identical in the database and the XML) we capture the table, row and column using a temporary *Touched* table (in the database or main memory). The Touched table has three columns; TableName, PrimaryKeyValue (the composite primary key), and ColumnName. When an update takes place, we check if the value has been updated/accepted before. If so, an exception is raised. If not, the update takes place and information about it is added to the Touched table.

5.5.3 Inferring a Plan for the Import

5.5.3.1 Insert and Update

We now consider how to do the actual work when inserting or updating from XML. Later we consider how to delete by means of an XML document. In order to reason on importability of the data of a concept, we build a *database model*, used for inferring database properties, and decide whether there is enough information to import the data and to infer an insertion order. A specific order may be required because of integrity constraints on the database. The database model holds information on the included tables and columns and their types. Furthermore, the model holds information on the primary keys of the included tables and links (foreign key constraints) between the tables of the concept. We have three types of links in the database model. *Hard links* represent foreign key constraints which are neither deferrable nor nullable; *semi-hard links* represent foreign key constraints which are not deferrable but nullable; *soft links* represent deferrable foreign key constraints.

A concept is viewed as an undirected *concept graph*, where nodes represent tables and edges represent the joins of the concept. Each edge is either an equijoin edge which follows the constraints of the database (represented as a solid line) or a non-equijoin edge or an equijoin edge which does not follow the constraints of the database (both represented as a dotted line). Figure 5.8 gives examples.

The *execution plan* determines the insertion order. Based on a concept and its database model, it is possible to build an execution plan to be used when importing.

The join types used in the concept, the columns joined and the structure of the database schema influence how to handle an insert or update. The data of a concept may be extracted from the database in many ways, some of which do not reflect the database constraints. For example, a concept may join on two columns not related by a database foreign key and may neglect another foreign key. Thus, data for a single data row may not always be consistent with the foreign key constraints, i.e., these are not fulfilled for the row.

For the import, we construct an insertion order which is a list of table lists. A table list shows tables which may be handled in the same run (parsing) through the

XML document, as the data rows are consistent with the database constraints. Thus, the length of the outer insertion order list is the required number of runs through the XML document.

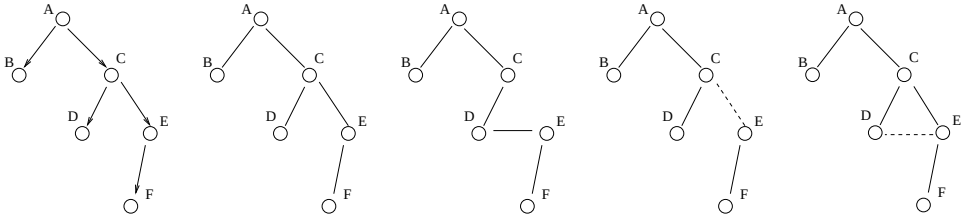


Figure 5.8: (a) Database model (b)-(e) Concept graphs

The database model in Figure 5.8(a) shows that table A has foreign keys to tables B and C, table C has foreign keys to tables D and E, and table E has a foreign key to table F. Figures 5.8(b)-(e) show the concept graphs for four different concepts using the database modeled in Figure 5.8(a).

The concept graph in Figure 5.8(b) shows that the data of each data row is guaranteed to be consistent with the database constraints, as the joins used in the export reflect these constraints and because each join is an equijoin. This is also the case for the Mini Market example in Figure 5.1. Figure 5.8(b) gives the insertion order $((F, B, D, E, C, A))$. The data from F is inserted before the data from E because the database model shows that the foreign key in E references F. In Figure 5.8(c), only equijoins are present, but the foreign key constraint from table C to E is not represented in the concept. Compared to the database model there is also an extra equijoin between the tables D and E. The missing equijoin between tables C and E means that in general we cannot insert the data rows at one time but must break the insertion into multiple phases. A possible insertion order is therefore $((B, F, D, E), (C, A))$. In Figure 5.8(d), all the constraints of the database model are fulfilled, except that there is a non-equijoin between tables C and E. This leads to the same situation as in Figure 5.8(c). In Figure 5.8(e), we get the insertion order $((B, F, D), (E, C, A))$, since D has an equijoin to E. We cannot continue with E in the first run since the D-E join might include a tuple of E, which does not fulfill the foreign key constraint between E and F.

So far, the database models have had no cycles. If cycles are present, we may break a cycle if it has at least one soft link or semi-hard link. A soft link may be deferred and a semi-hard link may be set to null first and updated to the correct value as the final step in the import. We refer to columns having pending updates as *postponed columns.*

Now, the execution plan holds an insertion order (the tables of the concept in a list of table lists) and a list of postponed columns. In the following, let an *independent table* be a table which is guaranteed to fulfill the constraints, i.e., does not have any outgoing links in the current database model. Algorithm 5.4 takes as input a concept c. In line 1, we build the database model and in line 2 we initialize the set of postponed columns to the empty set. Lines 3-4 remove all soft-links from the database model, i.e., edges representing deferrable constraints. In lines 5-6, we remove all semi-hard links from the database model, i.e., the deferrable and nullable constraints. The columns involved are added to the set of postponed columns. In lines 7-8, we check that there are no cycles in the database model. In this highly unlikely situation we are not able to continue, because there is a cycle of hard links. In line 9, we build the concept graph and in line 10 we initialize the insertion order list to the empty list. The while loop in lines 11-20 builds the insertion order list that consists of table lists. In line 12, the table list that can be inserted in one pass is initialized to the empty list. Lines 13-15 add to the table list, all tables that are joined by equijoins. These tables are removed from the database model. Lines 16-19 do the same for all independent tables. In line 20 the table list is prefixed to the insertion order list. Finally, in line 21 we reverse the insertion list and line 22 returns this list along with the set of postponed columns.

The importer uses the insertion order and handles one data row at a time. The insertion order shows how the importer should progress in the current run through the XML file.

5.5.3.2 Delete

We now describe an algorithm that handles deletion in database schemas which may be represented as directed acyclic graphs (DAGs) and schemas that hold cycles with cascade actions on all constraints in the cycle (termed *cascade cycles*). In addition, we consider modifications to the delete operation such that a larger set of database schemas can be handled.

As described in Definition 5.3.4, we delete a tuple from the database if there is a match on all values in the corresponding data in the XML document. When deleting, tuples that are referencing one or more of the tuples to be deleted may block the deletion. Even though a foreign key constraint is fulfilled in the data row, other tuples in the database may also reference a tuple d we want to delete. Thus, d cannot be deleted until all tuples that reference it are gone. This cannot happen before we see the last row in the derived table that d constitutes a part of. However, the derived table is denormalized and we do not know on beforehand which data row is the last to hold data from d. For this reason, we delete rows from lists of tables which are independent with regards to delete and foreign key constraints.

Algorithm 5.4 Building an execution plan

1: $dbm \leftarrow$ a database model for the concept c
2: $ppCols \leftarrow \emptyset$
3: **if** $commitInterval = \infty$ **then**
4: remove soft links from dbm
5: **if** cycles are present in dbm **then**
6: break the cycles by postponing a number of semi-hard foreign key columns, add them to $ppCols$
7: **if** cycles are still present in dbm **then**
8: Error - not importable (cycle of hard links exists)
9: $conceptGraph \leftarrow$ a concept graph of the concept
10: $iOrder \leftarrow ()$
11: **while** dbm has more nodes **do**
12: $tableList \leftarrow ()$
13: **while** dbm has an independent node n referenced by m where n and m are joined using an equijoin in $conceptGraph$ and n is not joined with other tables **do**
14: $tableList \leftarrow n :: tableList$
15: $dbm \leftarrow dbm$ without n
16: $indep \leftarrow$ independent nodes in dbm
17: **for** each node $node$ in $indep$ **do**
18: $tableList \leftarrow node :: tableList$
19: $dbm \leftarrow dbm$ without $node$
20: $iOrder \leftarrow reverse(tableList) :: iOrder$
21: $iOrder \leftarrow reverse(iOrder)$
22: **return** $(iOrder, ppCols)$

It is possible that the user has specified delete actions on foreign key constraints, such that a deletion causes a side effect. Delete actions can be defined on foreign key constraints and resolve constraint violations in case referenced tuples are deleted. Possible delete actions are *set null* (the foreign keys are set to null), *set default* (the foreign keys are set to a default value) and *cascade* (the referencing tuples are deleted).

The deletion order is very important. Consider a database schema where table A references table B. A tuple from B may only be deleted when no tuples in A reference the tuple in B. For efficiency reasons we do not want to query the database for referencing tuples for all tuples to delete. Instead we run through the XML twice. First deleting the data from A and then the data from B. Because of the definition of delete we may get a situation where tuples in A are updated as a side effect to

deletion in B such that we cannot delete them. This is the case if a set null or set default action is defined in the database such that deletion of a tuple in B has a side effect on tuples in A. If the action is cascading delete, the side effect does the job and one run suffices.

We use the database model for inferring a deletion order. The deletion order is a list of lists of tables. The inner lists show tables it is safe to delete from in the same run.

As when inserting, it is possible to specify a commit interval. If the commit interval is set to ∞ we may defer deferrable constraints. In this way, we may break some of the cycles in the database model.

In the following, we assume that the database schema can be represented as a DAG. When inferring a deletion order, actions have an impact on the deletion order.

In Figure 5.9(a), no actions are defined. We can use the order $((A), (B, C), (D, E, F, G))$.If the action is a cascading delete action in Figure 5.9(b), we may delete from A and Cin the same run since the action solves constraint violations. An order is therefore $((A, C), (B, F, G), (D, E))$. If the action is a set null action we cannot delete from C in the same run as A since a deletion in C may update tuples in A. This can have an impact on the tuples in A which are then not equal to the tuples read from the XML.

Assume that the database schema contains cascade cycles. We may delete data from such a cycle if all incoming links also have cascading delete actions. In such a situation we may still perform the delete operation. [61] provides an algorithm for inferring a deletion order in schemas without cycles or where the only cycles present are cascade cycles.

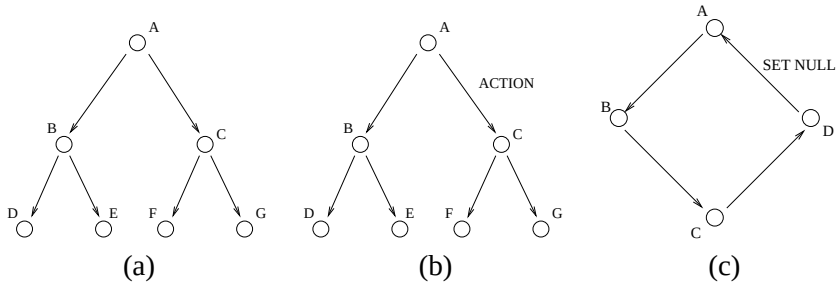


Figure 5.9: Schemas (a) DAG, no action (b) DAG, action (c) Cycle, action

As argued earlier, other delete actions invalidates the delete operation because of side effects that influence if a row should be deleted. Consider the cycle in Figure 5.9(c) where a schema with a cycle with a set null action is shown. We can break such a cycle if we delete from A and then proceed on the remaining graph (B, C, D). However, the side effect on the tuples of D and the definition of delete may cause a

Name	Range	Description
ID	$0, \ldots, r-1$	Primary key for the table.
ParentID	$0, \ldots, r-1$, null	Foreign key to ID. For each row, the value is ID -1 if ID mod $5 \neq 0$ Otherwise null.
GroupID	$0, \ldots, \lceil \frac{r}{5} \rceil - 1$	For each row, the value is $\lceil (\text{ID}+1)/5 \rceil - 1$
DLLevel	$0, \ldots, 4$	For each row, the value is ID mod 5
Random	$0, \ldots, 9$	Each row holds a random value
Fixed	c	For each row, the value is c

Table 5.1: Description of data in performance study

situation where we cannot delete tuples in D. If we change the delete operation to only consider equality of the primary keys, the cycle in Figure 5.9(c) may be broken such that the deletion can be performed correctly. The alternative solution can handle schemas with non-overlapping cycles (cascade cycles or at least one set null or set default action) but updates to the database are not retained which may be surprising to the user.

5.6 Performance Study

An implementation with approximately 15,000 lines of Java is done. The implementation is open source and is available from www.cs.aau.dk/∼chr/relaxml/ and www.relaxml.com. Performance tests have been carried out on a 2.6 GHz Pentium 4 with 1GB RAM, running SuSE Linux 9.1, PostgreSQL 8.0, and Java 1.4.2 SE. Every measurement is performed 5 times. The highest and lowest values are discarded and an average is computed using the middle three. The used test data and the test suites can be downloaded from the same places as the implementation.

The data is placed in a table with five integer columns and one varchar column: (ID, ParentID, GroupID, DLLevel, Random, Fixed). The values of the rows are as described in Table 5.1. For $r = 10$ this could result in the data in Table 5.2.

Export test 1 - Scalability in the number of rows This test exports all six columns. The data is exported to an XML structure as the following (but without unecessary spaces) where no grouping is used.

```
<Data>
   <GroupID value="[GroupID]">
      <Fixed value="[Fixed]">
         <ID>[ID]</ID>
```

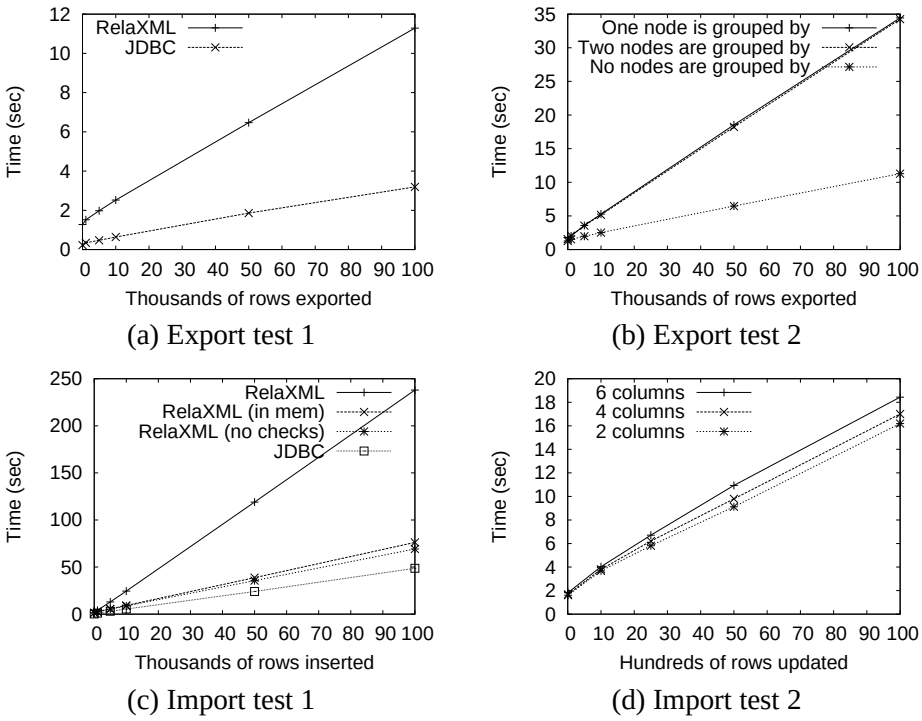
ID	ParentID	GroupID	DLLevel	Random	Fixed
0	null	0	0	3	c
1	0	0	1	8	c
2	1	0	2	2	c
3	2	0	3	7	c
4	3	0	4	1	c
5	null	1	0	4	c
6	5	1	1	0	c
7	6	1	2	5	c
8	7	1	3	9	c
9	8	1	4	6	c

Table 5.2: Example data

```
        <ParentID>[ParentID]</ParentID>
        <DLLevel>[DLLevel]</DLLevel>
        <Random>[Random]</Random>
      </Fixed>
   </GroupID>
   <GroupID ...>
      ...
   </GroupID>
   ...
</Data>
```

Figure 5.10(a) compares the running time of RELAXML with that of a specialized JDBC application that executes the SQL query corresponding to the used RELAXML concept. Both write the result set to an XML file, the structure of which has been hard-coded into the JDBC application. The results show that both RELAXML and the JDBC application scale linearly in the number of rows to export. From the slopes, it is seen that RELAXML handles on average 10.4 rows each millisecond (ms) whereas the JDBC application handles 37.5 rows each ms, i.e., the RELAXML overhead is 260%. This is a reasonable overhead given the flexibility and labor-savings of using RELAXML, especially taking into account that the XML documents used in web services are usually not very large.

Export test 2 - Scalability when grouping Here, the same data as in Export test 1 is exported, but now grouping is used. The data is grouped by one (GroupID) and two (GroupID and Fixed) nodes. The running time for no grouping, is the same for RELAXML in Export test 1. The results, in Figure 5.10(b), show that RELAXML also scales linearly in the number of rows when grouping. The performance suffers



(a) Export test 1 (b) Export test 2

(c) Import test 1 (d) Import test 2

Figure 5.10: Performance tests

when grouping is used, as one row takes approximately 3.3 times longer to export. This is as expected, since the use of grouping requires all the rows to be inserted into a temporary table in the database before they are sorted and then retrieved by the XML writer. The performance is the same when we are grouping by one and two nodes even though there is more sorting to do when grouping by two nodes. However, more data (30%) has to be written when we group by one node, as more tags are written since fewer elements are coalesced.

Export test 3 - Scalability in the number of dead links This test selects the rows where DLLevel = 4. Here, each selected row leads to four dead links which are resolved by RELAXML. The results are shown in Figure 5.11.

The running time of RELAXML does not scale linearly in the number of dead links resolved. This is expected since each time Algorithm 5.1 is invoked there will be more rows to search for dead links (leading to an approximately quadratic complexity). Further, the query gets more complicated to process as more OR clauses are added. Note that typical data sets will not contain so many dead links.

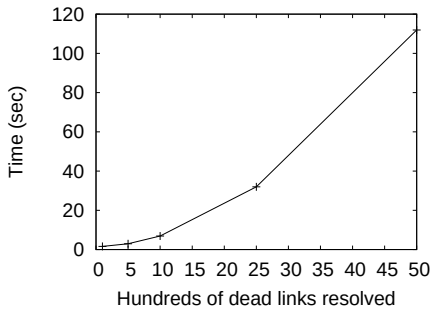


Figure 5.11: Performance test of resolving of dead links

Import test 1 - Scalability in the number rows to insert We now compare the time used by RELAXML for inserting with the time used for parsing the XML file with a SAX parser and inserting the data through JDBC prepared statements, checking that this will not lead to a primary key violation. Further, we consider the time used by RELAXML when the inconsistency checks are done in main memory or disabled. The data to insert originates from Export test 1. The table is emptied before the test is executed. The times used for inserting different numbers of rows are shown in Figure 5.10(c). The results show that both RELAXML and the JDBC application scale linearly. The average time to import a row using RELAXML is 2.38 ms. When checks for inconsistencies are performed entirely in main memory, RELAXML handles a row in 0.75 ms. If RELAXML does not check for inconsistencies, it handles a row in 0.67 ms, compared to 0.49 ms for using JDBC directly, i.e., the overhead from using RELAXML is only 37%.

Import test 2 - Scalability in the number rows to update We now focus on the scalability in the number of updates. We consider the impacts of updates to one column in rows from the table, varying the number of updated rows. When the XML document is processed, all the included rows have been updated. Only the column Fixed is updated, but the test has been performed with 2, 4, and 6 columns in the used concept. The results, in Figure 5.10(d), show that the running times are growing linearly in the number of rows after the data sets reach a certain size. The checks for inconsistencies are performed in main memory. More time is used when more columns are included, since more data has to be read from the XML document and more comparisons have to be performed. When 10,000 rows with 6 columns are included, it takes 1.8 ms to read a row and update it in the database. When 4 and 2 columns are included, it takes 1.7 ms and 1.6 ms, respectively.

In summary, we find that the overhead of RELAXML is very reasonable considering the flexibility, simplicity, and labor-savings of RELAXML compared to hard-coded applications. Further, to the best of our knowledge this is the first work to present a performance study of a general framework for bidirectional transfer of data between relations and XML documents.

5.7 Conclusion and Future Work

Motivated by the increasing exchange of relational data through XML based technologies such as web services, this chapter investigated automatic and effective bidirectional transfer between relational and XML data.

As the foundation, we proposed the notion of *concepts*, which are view-like mechanisms, for specifying the subset of data to export from a database to XML documents. Concepts support multiple inheritance and are therefore flexible to use. In addition, this allows specializations to be specified in an incremental fashion. The separation of concept from *structure definition* allows multiple XML representations of the same data. Further, the user-defined transformations allow changes to data that can be difficult to implement in SQL, e.g., ensure that parts of an export XML document are not altered. The specification of import and export ensured that data sets are self-contained such that they can be imported into an empty database without violating integrity constraints. Performance studies showed a reasonable overhead when exporting and importing compared to the equivalent hand-coded programs. This overhead is easily offset by the offered flexibility, simplicity, and labor-savings of RELAXML compared to hand-coded programs, e.g., in web-service applications.

There are a number of interesting directions of future research. Currently only inheritance between concepts is allowed. It would be interesting to allow aggregation such that the data from one concept can be included as a single element in another concept. It could also be investigated how to extend the approach to support XML documents with more freely defined structures, e.g, mixed content and irregular nesting structures. Another topic is the dead-link detection algorithm that possibly can be tuned by using SQL IN or BETWEEN statements instead of ORing as it is done now. Finally, it would be interesting to investigate whether the overhead compared to hand-coded applications can be avoided altogether by using the concept specification to auto-generate concept-specific Java code which is then just-in-time compiled before the execution.

Chapter 6

ETLDiff: A Semi-Automatic Framework for Regression Test of ETL Software

Modern software development methods such as Extreme Programming (XP) favor the use of frequently repeated tests, so-called regression tests, to catch new errors when software is updated or tuned, by checking that the software still produces the right results for a reference input. Regression testing is also very valuable for Extract–Transform–Load (ETL) software, as ETL software tends to be very complex and error-prone. However, regression testing of ETL software is currently cumbersome and requires large manual efforts. In this chapter, we describe a novel, easy–to–use, and efficient semi–automatic test framework for regression test of ETL software. By automatically analyzing the schema, the tool detects how tables are related, and uses this knowledge, along with optional user specifications, to determine exactly what data warehouse (DW) data should be identical across test ETL runs, leaving out change-prone values such as surrogate keys. The framework also provides tools for quickly detecting and displaying differences between the current ETL results and the reference results. In summary, manual work for test setup is reduced to a minimum, while still ensuring an efficient testing procedure.

6.1 Introduction

When software is changed, new errors may easily be introduced. To find introduced errors or new behaviors, modern software development methods like Extreme Pro-

gramming (XP) [10] favor so-called regression tests which are repeated for every change. After a change in the software, the tests can be used again and the actual results can be compared to the expected results.

A unit-testing tool like JUnit [55] is well-suited to use as a framework for such tests. In JUnit, the programmer can specify assertions that should be true at a specific point. If an assertion does not hold, the programmer will be informed about the failed assertion. In a framework like JUnit it is also very easy to re-run tests and automatically have the actual results compared to the expected results.

As is well-known in the data warehouse (DW) community, Extract–Transform–Load (ETL) software is both complex and error prone. For example, it is estimated that 80% of the development time for a DW project is spent on ETL development [57]. Further, ETL software may often be changed to increase performance, to handle changed or added data sources, and/or to use new software products. For these reasons, regression testing is essential to use. However, to the best of our knowledge, no prior work has dealt with regression testing for ETL software.

As a use case consider an enterprise with an ETL application that has been used for some time but that does not scale well enough to handle the data anymore. The enterprise's IT department therefore establishes a developer team to tune the ETL. The team tries out many different ideas to select the best options. Thus many different test versions of the ETL application are being produced and tested. For each of these versions, it is of course essential that it produces the same results in the DW as the old solution. To test for this criterion, the team does regression testing such that each new test version is being run and the results of the load are compared to the *reference results* produced by the old ETL application.

A general framework like JUnit is not suited for regression testing of the entire ETL process. Normally, JUnit and similar tools are used for small, well-defined parts or functions in the code. Further, it is more or less explicitly assumed that there is functional behavior, i.e., no side-effects, such that a function returns the same result each time it is given the same arguments. On the contrary, what should be tested for ETL software is the result of the entire ETL run or, in other words, the obtained side effects, not just individual function values. Although it is possible to test for side effects in JUnit, it is very difficult to specify the test cases since the database state, as argued in [22], should be regarded as a part of the input and output space. But even when the data is fixed in the input sources for the ETL, some things may change. For example, the order of fetched data rows may vary in the relational model. Additionally, attributes obtained from *sequences* may have different values in different runs. However, actual values for surrogate keys assigned values from sequences are not interesting, whereas it indeed is interesting how rows are "connected" with respect to primary key/foreign key pairs. More concretely, it is not interesting if an ID attribute A has been assigned the value 1 or the value 25

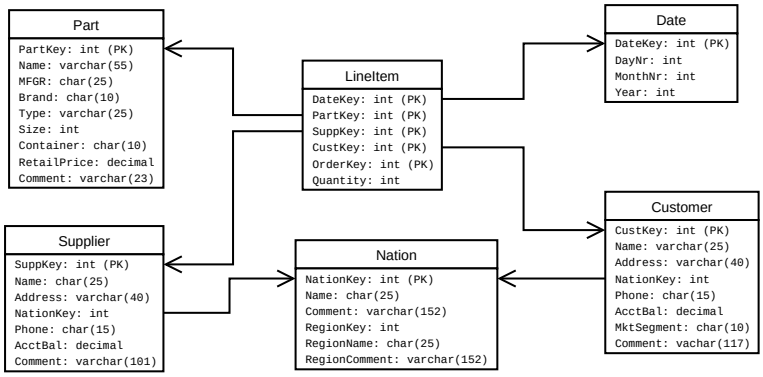


Figure 6.1: An example schema

from a sequence. What is important, is that any other attribute that is supposed to reference A has the correct value. Further, the results to compare from an ETL run have a highly complex structure as data in several tables has to be compared. This makes it very hard to specify the test manually in JUnit.

In this chapter, we present *ETLDiff* which is a semi-automatic framework for regression testing ETL software. This framework will, based on information obtained from the schema, suggest what data to compare between ETL runs. Optionally the user may also specify joins, tables, and columns to include/ignore in the comparison. ETLDiff can then generate the so-called *reference results*, an off-line copy of the DW content. Whenever the ETL software has been changed, the reference results can be compared with the current results, called the *test results*, and any differences will be pointed out. In the use case described above, the tuning team can thus use ETLDiff as a labor-saving regression testing tool.

Consider the example in Figure 6.1 which will be used as a running example in the rest of the chapter. The schema is for a DW based on source data taken from TPC-H [110].

Here we have a fact table, *LineItem*, and four dimension tables, *Date*, *Part*, *Supplier*, *Customer*, and an outrigger, *Nation*. The fact table has a degenerate dimension (*OrderKey*) and one measure. ETLDiff can automatically detect the six joins to perform and which columns to disregard in the comparison. ETLDiff will here make a join for each foreign key and then disregard the actual values of the columns involved in the joins.

To use the framework the user only has to specify A1) how to start the ETL software and A2) how to connect to the data warehouse, as shown in Figure 6.2. Apart from this, the framework can do the rest. Thus, the user can start to do regression testing in 5 minutes. Setting this up manually would require much more time. For

```
etlcmd='loaddw -f -x'
dbuser='tiger'
dbdriver='org.postgresql.Driver'
dburl='jdbc:postgresql://localhost/tpch'
```

Figure 6.2: Example configuration file for ETLDiff

each schema, the user would have to go through the following tasks: M1) write an SQL expression that joins the relevant tables and selects the columns to compare, M2) verify that the query is correct and includes everything needed in comparisons, M3) execute the query and write the result to a file, M4) write an application that can compare results and point out differences. Further, the user would have to go through the following tasks for each ETL version: M5) Run the new ETL software, M6) run the query from M1 again, M7) start the application from M4 to compare the results from M6 to the file from M3. Even though much of this could be automated, it would take even more work to set this up. Thus, to set up regression testing manually takes days instead of minutes.

The rest of this chapter is structured as follows. Section 6.2 describes the design and implementation of the ETLDiff framework. A performance study is presented in Section 6.3. Section 6.4 presents related work. Section 6.5 concludes and points to future work.

6.2 The Test Framework

In this section we present, how ETLDiff is designed and implemented. There are two basic parts of ETLDiff. A *test designer* and a *test executer*.

A test consists of all rows in the considered tables which are equi-joined accordingly to foreign keys. Thus the fact table is joined to each of the dimension tables. However, only some of the columns are used in the comparison. In the following it is explained how to select the data to compare.

6.2.1 Process Overview

ETLDiff's test designer makes a proposal about which data to include in a test. It does so by exploring the DW schema and building a *database model* of the schema (task 1). This model is used to build the so-called *join tree* (task 2a) which defines how to join the DW tables used in the test. When this is done, special care has to be taken when handling so-called *bridge tables* (task 2b). Since ETLDiff uses the tool

RELAXML [60] to export DW data to off-line XML files, certain files that define this process have to be generated as the last part of proposing a test (task 3). When executing a test, ETLDiff exports test results to a file (task 4). This file, with the newest content from the DW, is then compared to a file holding the reference results and differences are pointed out (task 5). In the following subsections, these tasks are explained.

6.2.2 Task 1: Exploring the DW Schema and Building a Database Model

To find the data to compare, ETLDiff builds a *database model* of the database schema. A database model represents tables and their columns, including foreign key relationships. The model is simply built based on metadata obtained through JDBC. By default, all tables and all their columns are included in the model. However, the user may specify a table name or just a prefix or suffix of names not to include. The user may also specify foreign keys that should be added to the model even though they are not declared in the database schema or may conversely specify specific foreign keys declared in the schema that should not be included in the model. For the DW example from Section 6.1, the built model would be similar to the schema shown in Figure 6.1 unless the user specified something else, e.g., to ignore the foreign key between *LineItem* and *Date*.

Next, ETLDiff has to find the columns to compare. In a DW, it is good practice to use surrogate keys not carrying any operational meaning [57, 58]. As previously argued, it is not important whether a surrogate key has the value 1 or the value 25 as long as attributes supposed to reference it have the correct value. For that reason, ETLDiff uses a heuristic where all foreign keys and the referenced keys in the model are left out from the data comparison unless the user has specified that they should be included. In the example from Section 6.1, this would mean that *OrderKey*, *PartKey*, *SuppKey*, *CustKey*, and *NationKey* would not be included in the comparison. The rest of the columns in the example would be used in the comparison. Also columns the user has explicitly chosen not to include will be disregarded. For example, it could be specified that *RegionKey* should not be compared in the running example.

6.2.3 Task 2a: Building a Join Tree

Consider again the running example. Since both *Supplier* and *Customer* reference the *Nation* outrigger, an instance of *Nation* should be joined to *Customer* and another instance of *Nation* should be joined to *Supplier*.

In more general terms, there must be an equi-join with an instance of a table for each foreign key referencing the table in the database model. This means the database model is converted into a tree, here called a *join tree*. Note that the database model

already can be seen as a directed graph where the nodes are the tables and the edges are the foreign keys between tables.

In the join tree, nodes represent table instances and edges represent foreign keys (technically, the edges are marked with the names of the key columns). For a star schema, the root of the tree represents the fact table and the nodes at level 1 represent the dimension tables. Outriggers are represented at level 2 as children of the nodes representing the referencing dimension tables. For a snowflake schema, the join tree will have a level for each level in the dimension hierarchy. The join tree for the running example is shown in Figure 6.3 (not showing the marks on the edges).

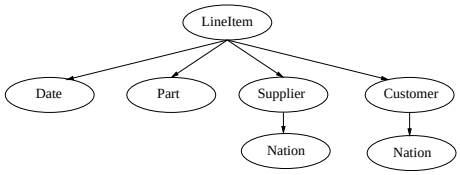


Figure 6.3: A join tree for the running example

To convert the database model into a join tree, we use Algorithm 6.1, BuildJoinTree, which is explained in the following. To avoid infinite recursion when AddTreeNodesDF is called, we require that the database model does not contain any cycles, i.e., we require that the database model when viewed as graph is a directed acyclic graph (DAG). This is checked in l 1–2 of the algorithm. Note that this requirement holds for both star and snow-flake schemas.

In l 3 the array $visited$ is initialized. In l 4, the algorithm tries to guess the fact table unless the user explicitly has specified the fact table. To do this, the algorithm considers nodes in the database model with in-degree 0. Such nodes usually represent fact tables. However, they may also represent the special case of *bridge tables* [57,58] which will be explained later. To find the fact table, the algorithm looks among the found nodes for the node with maximal out-degree. If there are more such nodes, the first of them is chosen, and the user is warned about the ambiguity. Another heuristic would be to consider the number of rows in the represented tables. The one with the largest number of rows is more likely to be the fact table. We let f denote the node in the database model that represents the fact table. In the join tree, the root is representing the fact table (l 5). The recursive algorithm AddTreeNodesDF (not shown) visits nodes in a depth-first order in the database model from the node representing the fact table (l 7–8). When a node representing table t in the database model is visited from node n, a new node representing t is added in the join tree as a child of the latest added node representing n. This will also set $visited[t]$ to true. Note that

Algorithm 6.1 BuildJoinTree

```
1: if database model has cycles then
2:    raise an error
3: set visited[t] = false for each node t in the database model
4: f ← GuessFactTable()
5: root ← TreeNode(f)
6: visited[f] ← true
7: for each node t adjacent to f in the database model do
8:    AddTreeNodesDF(root, t)
9: // Find bridge tables and what is reachable from them
10: changed ← true
11: while changed do
12:    changed ← false
13:    for each table node t in the database model where visited[t] = false do
14:       oldVisited ← visited
15:       for each node s adjacent to t in the database model do
16:          if oldVisited[s] then
17:             // Before this part, t had not been visited, but s which is referenced
18:             // by t had, so t should be included as if there were an edge (s, t)
19:             for each join tree node x representing table s do
20:                Remove edge (t, s) from database model // Don't come back to s
21:                AddTreeNodesDF(x, t) // Modifies visited
22:                Add edge (t, s) to database model again
23:             changed ← true
```

AddTreeNodesDF *will* visit an adjacent node even though that node has been visited before. This for example happens for *Nation* in the running example.

For the running example, the nodes in the database model are visited in the order *LineItem, Date, Part, Supplier, Nation, Customer, Nation*. Only the already explained part of the algorithm is needed for that. For some database models this part is, however, not enough, as explained next.

6.2.4 Task 2b: Handling Bridge Tables

In the depth-first search only those nodes reachable from f will be found. In fact we are only interested in finding the nodes that are connected to f when we ignore the direction of edges. Other nodes that are unvisited after the algorithm terminates represent tables that hold data that is not related to the data in the fact table. However, nodes may be connected to f when we ignore directions of edges but not when directions are taken into consideration. Imagine that the example DW should be able

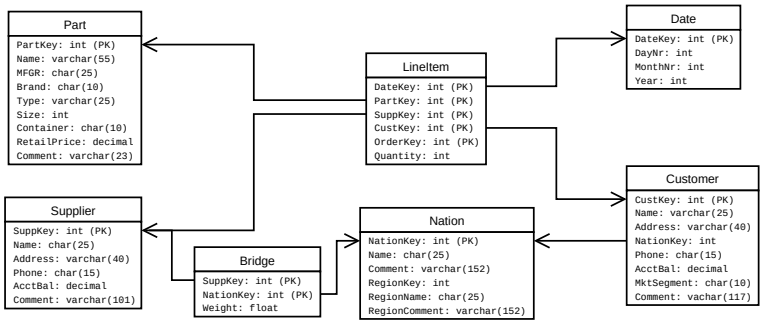


Figure 6.4: The example schema extended with a bridge table between Supplier and Nation

to represent that a supplier is located in many nations. To do this we would use a bridge table [57,58] as shown in Figure 6.4. A bridge table and nodes reachable from the bridge table should also be visited when the join tree is being built. Before terminating, the algorithm therefore has to look for unvisited nodes that have an edge to a visited node (l 13 and 15–16). If such an edge is found, it is "turned around" temporarily such that the depth-first visit will go to the unvisited, but connected node. To do this, the edge is removed from the database model (l 20), and a call to AddTreeNodesDF is then made (l 21) as if the edge had the opposite direction. Since the edge is removed from the model, this call will not come back to the already visited node. After the call, the edge is recreated (l 22). Before the edge is turned around, it is necessary to make a copy of the $visited$ array. The reason is that the algorithm otherwise could risk to find an unvisited node u where the visited node v and the unvisited node w are adjacent to u. The edge (u, v) could then be turned around and the depth-first visit could visit u and w, before (u, v) was recreated. But when w (which is adjacent to u) then was considered, it would be visited and the edge (u, w) would be turned around and too many nodes would be added. This situation does not occur when an unmodified copy ($oldVisited$) of $visited$ is used.

6.2.5 Task 3: Generating Data-Defining Files

ETLDiff uses RELAXML [60] for writing XML files. Proposing a test thus includes generating a so-called *concept* which defines what data RELAXML should export and a so-called *structure definition* which defines the structure of the XML. A concept can inherit from other concepts.

When the join tree has been built, the data-defining concept can be built. In RELAXML a table can only be used once in a single concept. However, it might be necessary to include data from a table several times as explained above. When this is

the case, ETLDiff can exploit RELAXML's concept inheritance. A simple concept is made for each node in the join tree. The concept simply selects all data in the table represented by the node. An enclosing concept that inherits (i.e., "uses the data") from all these simple concepts is then defined. The results of the different concepts are joined as dictated by the join tree. The enclosing concept will also disregard the columns that should not be considered, e.g, dummy keys. In the running example, the final data corresponds to all the columns except those participating in foreign key pairs. The raw data is computed by the DBMS.

After the concepts have been created, a structure definition is created. ETLDiff uses sorting and *grouping* such that similar XML elements are coalesced to make the resulting XML smaller (see [60]). If a supplier for example supplies many parts, it is enough to list the information about supplier once and then below that list the information about the different parts. Without grouping, the information on the supplier would be repeated for each different part it supplies. The use of grouping and sorting means that the order of the XML is known such that it is easy and efficient to compare the two XML documents.

6.2.6 Task 4: Exporting DW Data to Files

The concepts and the structure definition are then used by RELAXML when it generates the files holding the reference results and the test results. Based on the concept, RELAXML generates SQL to extract data from the DW and based on the structure definition, this data is written to an XML file. Since the data sets potentially can be very large, it is possible to specify that the output should be compressed using gzip while it is written.

6.2.7 Task 5: Comparing Data

When comparing data, there should be two data sets to consider. The desired result of an ETL run (the *reference results*) and the current result (the *test results*). ETLDiff thus performs two tasks when running a test: 1) Export data from the DW, and 2) compare the test results to the reference results and point out any differences found. ETLDiff can output information about differences to the console or to tables in a window as shown in Figure 6.5. The window has two tabs. In the first tab there is a table showing all the rows missing in the test results and in the second tab there is a table showing all the extra rows in the test results.

When comparing test results to reference results, two data sets are read from two XML files. These XML files are read using SAX [99]. Each of the SAX parsers is running as a separate thread. Each thread reads XML and regenerates rows as they were in the join result that was written to XML. In this way it is possible to compare the files part by part with a very small main-memory usage. Only two rows from each

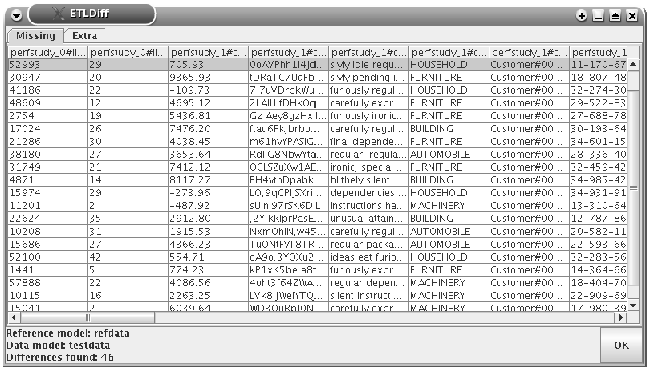


Figure 6.5: Window presenting differences between test results and reference results

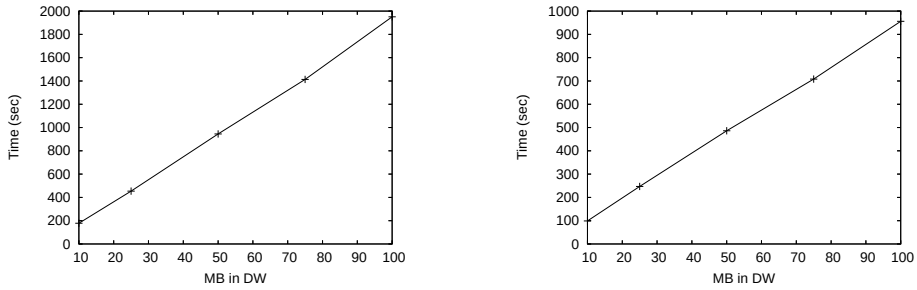
of the join results have to be in memory at a given time (each thread may, however, cache a number of rows). So for most use cases, the size of the data in main-memory is measured in kilobytes. Since sorting is used before the XML is written, it is easy to compare data from the XML files row by row.

6.3 Performance Results

A prototype[1] of ETLDiff has been implemented in Java 5.0. Further, RELAXML has been ported to Java 5.0, given new functionality, and performance-improved in a way that has speeded up the XML writing significantly. In this section, we present a performance study of the implemented prototype. The test machine is a 2.6 GHz Pentium 4 with 1GB RAM running openSuse 10.0, PostgreSQL 8.1, and Java 1.5.0 SE.

In the performance study, the DW from the running example has been used. ETLDiff has automatically proposed the test (this took 1.5 seconds). The data used originates from TPC-H's dbgen tool. Data sets with different sizes (10MB, 25MB, 50MB, 75MB, 100MB) have been loaded and a data set (which either could be test results or reference results) has been generated by ETLDiff. The resulting running times are plotted in Figure 6.6(a). The shown numbers indicate the total amount of time spent, including the time used by the DBMS to compute the join result. Notice that the data sets generated by ETLDiff contain redundancy and thus are much bigger than the raw data (10MB in the DW results in 129MB XML before compression and 10MB after compression).

[1]The source is publicly available from `relaxml.com/etldiff/`.



(a) Generating data sets (b) Comparing data sets

Figure 6.6: Running times

Further, the created test results have been compared to identical reference results. This is the worst case for equally sized data sets since it requires all data to be compared. The running times for the comparisons are plotted in Figure 6.6(b).

As is seen from the graphs, ETLDiff scales linearly in the size of the data, both when generating and comparing results. When generating, 19.7 seconds are used for each MB of base data and when comparing 9.5 seconds are used. This is efficient enough to be used for regression tests. In typical uses, one would have a data set for testing that is relatively small (i.e., often less than 100MB). The purpose of ETLDiff is to do regression testing to find newly introduced errors, not to do performance testing where much larger data sets are used. When regression testing, it is typically the case, that a single test case should be relatively fast to execute or that many test cases can be executed during a night. Thus, a test case should be small enough to be easy to work with but represent all special and normal cases that the ETL software should be able to handle.

6.4 Related Work

As previously mentioned, we believe that this work is the first framework for regression testing ETL tools. Daou *et al.* [29] describe regression testing of Oracle applications written in PL/SQL. The test cases to re-run are supposed to be automatically found by the described solution. The method used may, however, omit test cases that could reveal bugs [116]. In a recent paper [116], Willmor and Embury propose two new methods for regression test selection. The regression test selection solutions [29, 116] are closer to traditional combined unit and regression testing where there exist many manually specified tests that cover different parts of the code. In the present chapter, the result of the entire ETL run is being tested, but in a way that

ignores values in surrogate keys that can change between different runs without this indicates an error. Further, the test is designed automatically.

JUnit [55] is the de-facto standard for unit testing and has inspired many other unit-testing tools. In JUnit, it is assumed that the individual test cases are independent. Christensen *et al.* [23] argue why this should not hold for software that stores data in a database. They also propose a unit-test framework that allows and exploits structural dependencies to reduce coding efforts and execution times. The work is taking side-effects into consideration (such that a test can depend on the side-effect of another) but is still considering the individual functions of the tested program, not the entire result as the present chapter does. The main difference is that the solution in the present chapter automatically designs the test and is specialized for DWs.

DbUnit [32] is an interesting test framework extending JUnit for database applications. DbUnit can put the database in a known start state before any test run. Further, DbUnit can export database data to XML and import data from XML into the database. With respect to that, DbUnit has some similarities with RELAXML [60] used to write ETLDiff's XML. DbUnit can also compare if two tables or XML data sets are identical, also if specific columns are ignored. In that, it is related to the core functionality of ETLDiff. However, ETLDiff is automatic whereas DbUnit due to its unit test purposes requires some programming. Like in JUnit, the programmer has to program the test case and define the pass criterion for the test. This involves inheriting from a predefined class and defining the test methods. When using ETLDiff, the test case is automatically inferred. Another difference is that ETLDiff automatically will perform correct joins – also when disregarding the join columns in the value comparisons. Columns to ignore must be specified in DbUnit whereas in ETLDiff they are found automatically. A key feature of ETLDiff is that it uses the DW semantic to automate the tests.

One paper [52] considers the problem of discovering dimensional DW schemas (fact tables, measures, dimensions with hierarchies) in non-DW schemas. This is somewhat related to our problem of building a join tree, but as we can assume a DW schema and do not want to find hierarchies or measures, but only join connections, the algorithm in the present chapter is much more efficient. Addtionally, the solution in [52] does not handle bridge tables.

Industrial ETL tools like Informatica Powercenter, IBM Datastage, and Microsoft SQL Server Integration Services (SSIS) offer nice facilities for debugging ETL flows and allow ETL developers to use the testing facilities in Visual Studio, but have no specific support for ETL regression testing, and do not generate the test automatically, as we do.

In the implementation of ETLDiff, the data to compare is written to XML files in a way that allows for memory-efficient and fast processing. This means that when ETLDiff is comparing data sets, it actually compares data read from XML docu-

ments. Much work has been done in this area, see [25, 92] for surveys. Since the XML structure allows for a fast and memory-efficient comparison, ETLDiff uses its own comparison algorithm to be more efficient than general purpose tools.

6.5 Conclusion and Future Work

Motivated by the complexity of ETL software, this chapter considered how to do regression testing of ETL software. It was proposed to consider the result of the entire ETL run and not just the different functions of the ETL software. The semi-automatic framework ETLDiff proposed in the chapter can explore a data warehouse schema and detect how tables are connected. Based on this, it proposes how to join tables and what data to consider when comparing *test results* from a new ETL run to the *reference results*. It only takes 5 minutes to start using ETLDiff. The user only has to specify how to start the ETL and how to connect to the DW before he can start using ETLDiff. To setup such regression testing manually is a cumbersome task to code and requires a lot of time.

Performance studies of ETLDiff showed a good performance, both when extracting data to compare from the DW and when performing the actual comparison between the data in the DW and the so-called reference results. In typical uses, less than 100MB data will be used for testing purposes, and this can be handled in less than an hour on a typical desktop PC.

There are many interesting directions for future work. The structure definitions could be optimized with respect to group by such that the resulting XML gets as small as possible. The framework could also be extended to cover other test types, for example audit tests where the source data sets and the loaded DW data set are compared.

Chapter 7

RiTE: Providing On-Demand Data for Right-Time Data Warehousing

Data warehouses (DWs) have traditionally been loaded with data at regular time intervals, e.g., monthly, weekly, or daily, using fast *bulk loading* techniques. Recently, the trend is to insert all (or only some) new source data very quickly into DWs, called *near-realtime* DWs (*right-time DWs*). This is done using regular INSERT statements, resulting in too low insert speeds. There is thus a great need for a solution that makes inserted data available quickly, while still providing bulk-load insert speeds. This chapter presents *RiTE* ("Right-Time ETL"), a middleware system that provides exactly that. A data producer (ETL) can insert data that becomes available to data consumers *on demand*. RiTE includes an innovative main-memory based *catalyst* that provides fast storage and offers concurrency control. A number of policies controlling the bulk movement of data based on user requirements for persistency, availability, freshness, etc. are supported. The system works transparently to both producer and consumers. The system is integrated with an open source DBMS, and experiments show that it provides "the best of both worlds", i.e., INSERT-like data availability, but with bulk-load speeds (up to 10 times faster).

7.1 Introduction

Data warehouses (DWs) [58] have traditionally been loaded with data at regular time intervals, e.g., monthly, weekly, or daily. Here, fast *bulk loading* techniques have

typically been used in order to obtain sufficiently high insert speeds for the huge data volumes. In recent years, there has been an increasing demand for having very fresh data in DWs. Thus, new or updated data from the operational source systems has been inserted very quickly (within seconds or minutes) into the DWs, which are commonly referred to as "*near-realtime DWs*". A more sophisticated approach acknowledges that some data needs to be very fresh, while other data may be less fresh, and thus, based on the freshness needs, inserts data at the "right time" into the DWs, referred to as "*right-time DWs*". Bulk-loading techniques are only efficient for relatively large batches of data, and are thus not feasible for the single/few row "trickle feeds" used in the latter types of DWs. Thus, these have had to revert to classical OLTP-style inserts, using regular INSERT statements executed in small transactions. But here the unavoidable problem is that the insert speed is not high enough (often an order of magnitude lower than bulk loading).

There is thus a great need for a solution that makes inserted data available quickly, while still providing bulk-load insert speeds. A lot of work has been done on supporting *read-optimized* DWs, e.g. special multidimensional index structures, OLAP servers, etc. It is, however, equally necessary to have *write-optimized system* "before" the DW. Thus, we need a solution to asynchronously propagate data from sources to the DW (under some consistency constraints). Such a solution should strike the right batch size between the two extreme forms (bulk versus single row) and find the right time to move "micro batches" of data within the system. We note that data must be inserted at the latest *when*, but not necessarily *before*, it is needed, i.e., data should be available only on-demand. There is also a need to decouple source systems and the DW.

This chapter presents *RiTE* ("Right-Time ETL"), a middleware system that provides exactly such a solution. RiTE allows a data producer to continuously insert data into a DW at bulk-load speed, but such that data *consumers* (DW clients executing queries) get access to fresh data. To do this, RiTE takes advantage of a number of special characteristics of DW systems. RiTE is thus targeted at supporting one *producer* (the ETL program) doing many INSERTs with low persistency requirements (persistency can be guaranteed if needed). RiTE includes an innovative main-memory based *catalyst* that, like a chemical catalyst, enables the insert process to be performed faster and with less effort. RiTE supports a number of policies controlling the bulk movement of data based on user requirements for persistency, availability, freshness, as well as elapsed time and CPU load. Using RiTE is transparent and requires only very few changes to producer and consumer code, in most cases only the few lines establishing database connections have to be changed.

Figure 7.1(a) shows a classical DW system with source systems, a producer, a DW, and consumers. The black boxes show database drivers, e.g., JDBC [108]. Figure 7.1(b) shows the architecture for the same system using RiTE, with the catalyst

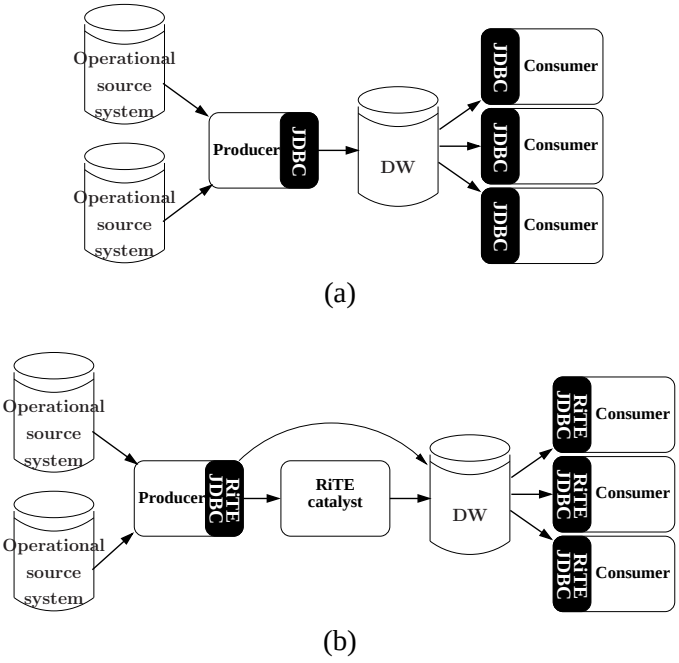


Figure 7.1: Architectures for (a) a classical system and (b) a system using RiTE

and specialized database drivers. The catalyst holds data in main memory but ensures that data is transparently available to the consumers. Data from the producer can then float to the DW either via the catalyst or directly.

Performance studies of the PostgreSQL-based prototype shows that RiTE improves insert time by up to an order of magnitude. Rows are transparently read from the RiTE catalyst with only a small overhead. Thus, RiTE provides INSERT-like data availability, but with bulk-load speeds.

The remainder of the chapter is structured as follows. Section 7.2 describes RiTE from a user perspective. Section 7.3 describes the producer database driver. Section 7.4 describes the catalyst. Section 7.5 describes the table function and the consumer database drivers. Section 7.6 presents experimental results. Section 7.7 presents related work and Section 7.8 concludes and points to future work.

7.2 User-Oriented Operations

We now give short, informal introductions to the operations that are treated specially by the RiTE package. These operations and other operations used internally by RiTE are all exemplified and described in details in the following sections. Note that other

classical database operations that are not handled specially by RiTE can still be performed.

Producer Operations The two producer operations *insert* and *commit* are handled specially by RiTE. From the user's point of view, *insert* operations work as normal inserts but are faster. Behind the scenes, RiTE temporarily keeps the inserted values locally at the producer side and later moves them towards the DW in bulk. The strategy about when to move data in bulk is based on different policies that are explained later. It is, however, done such that the data always is available from the DW when it is needed for querying.

The *commit* operation makes inserted data available for consumers. But when using RiTE, the user decides if committed data is written to the DW's tables. If this is done, the commit is called a *materialization*. If the user does not have strict persistency requirements (e.g., if the data can be re-extracted from the sources), it is also possible to commit the data without doing a materialization which then can be done later. This is faster, but still makes the data available for consumers. Such a commit can be done in different ways that affect when the bulk moving of data takes place.

Consumer Operations For a consumer, there are also two operations that are handled specially: *read* and *ensure accuracy*. From the user's point of view, a *read* is done by using SELECT. Behind the scenes, transparently to the user, the read is not necessarily just a read from tables in the DW.

The only new operation introduced by RiTE is *ensure accuracy*. This is relevant for a consumer that does not necessarily need data that is as fresh as possible and thus can help the system to get a better performance. For example, it may be acceptable for a daily status report to consider all sales data that existed 10 minutes ago but not newer data. By using the ensure accuracy operation, the consumer is guaranteed that it at least sees the data that existed 10 minutes ago.

7.3 Producer Side

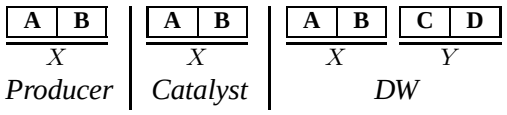
In this section, the specialized database driver for the producer is described.

Setup The RiTE producer driver is defined by an extension of the standard Java JDBC Connection interface. That means that to start using it from an existing Java application, only the lines where the connection to the database is made must be changed. The driver must be told which of the DW's tables the catalyst provides intermediate storage for (so-called *memory tables*). Inserts to these tables are then

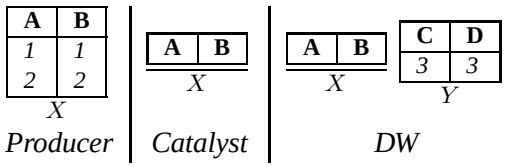
handled by the driver. Statements not handled specially by the RiTE driver are executed via a traditional JDBC Connection implementation.

Insert When a prepared statement is made, the driver detects if the statement inserts scalar values into a memory table. If so, the driver takes the values to insert from the statement when this is executed and stores them in a local buffer.

Example 7.3.1 (Insert) *Consider an example where the DW has two (empty) tables, $X(A, B)$ and $Y(C, D)$. RiTE is used such that a memory table is made for X. (This setup is used as a running example in the chapter.) Now, assume that the producer code with prepared statements inserts the rows $(1, 1)$ and $(2, 2)$ into X and $(3, 3)$ into Y. Before these inserts, the system has the following state where the local buffer for X is shown to the left, the catalyst's memory table for X in the middle and the DW's tables (from now on referred to as the* DW tables*) to the right. A double line in the bottom of a table shows that the table is empty.*

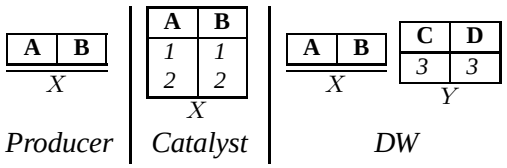


After the inserts, the system has the state shown below where the two new X rows are held in the local buffer and the new Y row is in the DW table Y.



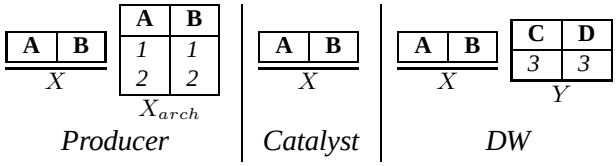
Flush The new rows from the prepared statement remain in the producer driver's buffer until a commit operation is done by the producer or optionally until the producer executes a query that should consider (uncommitted) data inserted by the producer itself. The held rows are then *flushed* to the catalyst (not the DW) in a bulk operation.

Example 7.3.2 (Flush) *Consider again the state obtained in Example 7.3.1 and assume that the producer commits the data such that a flush is initiated. This results in a state where the X rows have migrated to the catalyst.*



Lazy Commit It is also possible for the producer driver to keep rows locally *after* a commit whenever a *policy* defines to do so. When committed data is not flushed immediately, we have a *lazy commit*. When a lazy commit appears, the producer driver records the *commit time* at which commit operation was invoked and places all rows in the buffer in an *archive* which holds committed, but not flushed, rows. The archive is flushed later as explained below.

Example 7.3.3 (Lazy commit) *Consider again the state obtained in Example 7.3.1. If the producer performs a lazy commit, we get the following state where X_{arch} is an archive.*



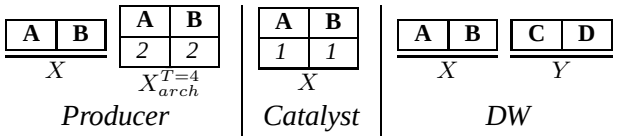
Compare this to the state obtained in Example 7.3.2. In the current example, the X rows are not migrating to the catalyst but remain on the producer side. After a flush is performed, the state resembles the situation of Example 7.3.2.

Requests for Data It is possible for the producer driver at the same time to have several archives with different commit times. These archives hold committed data that eventually should be flushed. At the latest, the flush is done when the connection to the DW is closed, but it may also happen before. When one or more archives exist, the producer driver sets up a background thread that listens for requests for data from the catalyst. As will be explained later, such a request occurs because a consumer has a demand for fresh data. The catalyst might ask only for parts of the archived data in which case only the requested parts are flushed. The recorded commit times are used to decide which parts to flush.

Example 7.3.4 (Request for data) *Consider again the running example and assume that lazy commits are used for the following sequence of events. The numbers shown to the left are (abstract) time stamps. Before the shown events, nothing has happened.*

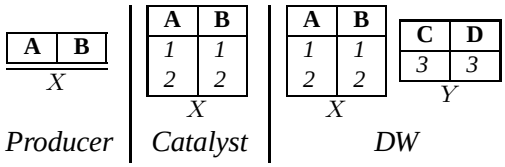
1. *The row $r = (1, 1)$ is inserted into X by the producer.*
2. *The producer commits, resulting in the archive $X_{arch}^{T=2}$ for X. This archive holds the row r.*
3. *The row $s = (2, 2)$ is inserted into X by the producer.*
4. *The producer commits. This results in that the archive $X_{arch}^{T=4}$ is made for X. This archive holds the row s.*

5. *The consumer requests the catalyst to hold data for X that is maximally 2 time units old. This means that the catalyst should at least hold the data committed at time $5 - 2 = 3$. To fulfill this, the catalyst sends a request for data to the producer. The producer then flushes the data in $X_{arch}^{T=2}$ (the only archive with data committed at time 3). Row r (committed at time 2) is then available from the catalyst, whereas row s (committed at time 4) is not. This gives the state shown below.*



Materialize Data from the archives is also flushed when the producer wishes to *materialize* the rows such that they are written to the DW tables. This is done to make the rows reach their final target (the DW table), to make space for other rows in the catalyst, and to guarantee persistency. Persistency is not guaranteed when rows are stored by the catalyst. In case of a crash, the rows in the catalyst will be lost. Recall that in typical DW environments this is not a problem since the data can be reloaded from the operational systems. When rows on the other hand have been materialized, the usual persistency guarantees given by the DW DBMS apply. Note that the producer thus controls the persistency guarantee since the catalyst does not do "implicit" materializations. To make materialization possible, the RiTE producer driver extends JDBC's Connection class with the method `commit(boolean)` which performs a commit operation and where the argument decides whether the rows should be materialized to the DW tables before the commit operation is performed in the DW. To make the rows ready for materialization, the producer driver first has to transfer them to the catalyst. Note that since a materialization only happens together with a commit operation, data held in the producer driver's local buffer is flushed at the same time.

Example 7.3.5 (Materialization) *Assume that the state is as obtained in Example 7.3.2. A materialization then gives the following state where the X rows are inserted into the DW.*



Note that the rows are still present in the catalyst after the materialization. However, it is automatically ensured that a consumer only sees each row instance once (this is explained in Section 7.5). When space is needed, the now materialized rows will eventually be deleted from the catalyst.

Policies Finally, data in archives is flushed when a *policy* has defined that it is time to do so. A policy is simply a function that returns a Boolean value. When the return value of the policy is `true`, the rows are flushed and vice versa. The producer invokes the policy and checks the return value at regular user-definable intervals. By using policies, it is for example possible to make the producer less intrusive on busy systems by considering the load average. A possible policy is thus only to flush if the load average for the last minute has been below 80% or if 10 minutes have passed since the last flush.

The RiTE package includes policies 1) for flushing immediately after a commit (this is the default), 2) for waiting as long as possible, i.e, only flush on-demand, and 3) for load-aware policy-based flushing when the load average is below some percentage or a certain time interval has passed since the last flush. Further, an interface that the user can implement to define her own policies is included. The interface has two functions: One for the policy itself, i.e., a function returning a Boolean value, and one used to inform the implementation that the data has been flushed for another reason, e.g., a request for data from the catalyst.

To start using lazy commit with a given policy, the user only has to define which policy to use. Thus, it only requires one line of code to start using a policy. The rest is handled transparently by the RiTE drivers.

The minmax Table When rows are flushed to the catalyst, the catalyst implicitly assigns *row IDs* to the rows and returns the maximal assigned row ID to the producer driver. The producer driver then updates a special metadata table, called the *minmax table*, in the DW. The minmax table holds data about the minimal and maximal row ID for rows that a consumer should get from the catalyst. Note that these row IDs are handled completely transparently by the RiTE software and are never seen by the producer or consumer code. So after a flush, rows with new row IDs are available and the information about the maximal available row ID is updated. As explained later, this only affects later consumer queries. Already running queries are unaffected and will not see the new rows that were committed after they started.

After a materialization, the producer driver similarly updates the information about the minimal row ID of rows that a new consumer query should get from the catalyst. The reason is that rows with lower row IDs now, after the materialization, have migrated to the tables in the DW.

Example 7.3.6 (The minmax table) *Consider again Example 7.3.2 where data was flushed. Assume that the row* $(1, 1)$ *is assigned row ID 1 and the row* $(2, 2)$ *is assigned row ID 2. After the flush, the minmax table has the content shown to the left below. After the materialization in Example 7.3.5, it has the content shown to the right.*

min	max
1	*2*

After Ex. 7.3.2

min	max
3	*2*

After Ex. 7.3.5

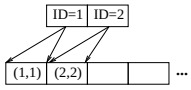
Note that after the materialization, the minmax table tells that consumers should get the empty set of rows from the catalyst since no row has an ID such that both ID ≤ 2 and ID ≥ 3 hold. The consumers should now get the rows from the DW table instead.

7.4 Catalyst Side

We now describe the catalyst. The purpose of the catalyst is to provide fast, intermediate storage for data. It does so by storing rows in main memory. It can serve one producer driver and many consumer drivers and their table functions at the same time. Note that the consumer driver itself does not fetch rows. Instead it (transparently to the user) informs the catalyst about which rows should be readable by a table function. A table function is the remedy that makes rows accessible in the DW. The catalyst is independent of the used DBMS as its sole functions are to 1) store rows for a producer, 2) deliver them to a table function, and 3) delete them when they are marked as unused (i.e., no consumer currently uses them and they have been materialized).

The Row Index The catalyst allocates a user-adjustable amount of memory for each memory table and uses this to store the memory table's rows. Whenever a producer driver adds rows, the rows are implicitly assigned row IDs by the catalyst. All row IDs are taken from the same sequence such that there are no duplicates among row IDs for different memory tables. The catalyst maintains a *row index* that is used to map between row IDs and start and end positions for the data of the rows. The row IDs are only stored in the row index, not together with the data of the rows.

Example 7.4.1 (The row index) *Consider again Example 7.3.2 and assume again that the row $(1, 1)$ is assigned the row ID 1 and the row $(2, 2)$ the row ID 2. Then the row index will be as shown to the right. (Note that although the row index here is shown as a list, a tree-based index is used in the implementation.)*



When a table function reads data, it gives the minimal and maximal needed row IDs (recall that these were made available in the minmax table by the producer driver). By using the row index, it is then very easy for the catalyst to find the chunk of memory to transfer to the table function.

The Time Index When a producer driver adds rows, it must tell the catalyst when the rows were committed at the producer side. For each memory table, the catalyst maintains a *time index* that for a commit time t maps to the row ID of the last row that was committed at time t. When a producer driver adds rows that are not yet committed (this is an option for a producer that needs to query its own uncommitted data), it gives them the special time stamp ∞.

Example 7.4.2 (The time index) *Consider again inserts into the memory table X in the running example and assume that the rows r_1 and r_2 are committed at time t_1 and the rows r_3 and r_4 are committed at time t_2. Assume that the row r_n gets the row ID n. The time index τ is then a partial function from time stamps to row IDs such that $\tau(t_1) = 2$ and $\tau(t_2) = 4$.*

A producer driver must transfer rows in a way where for a single memory table, all rows that were committed at time t_1 are flushed in one operation and before rows committed at time t_2 for $t_1 < t_2$. It therefore holds that when a producer driver adds uncommitted rows, it must already have added all its committed rows since they have commit times less than ∞. On the other hand, when new rows with a time stamp $t \neq \infty$ are added, all rows with the time stamp ∞ can implicitly be assumed to also be covered by this new commit and can have their time stamp updated to t. In case of a rollback, the catalyst simply has to discard all rows with the time stamp ∞. The chunk of memory that holds these rows is easy to identify by using the time and row indexes.

Ensuring accuracy. A consumer can tell the catalyst to ensure that it holds the data with a certain accuracy (i.e., the data that was committed by the producer a certain time interval ago) for a subset of the memory tables. The default is that the catalyst should have all data, but with a one-line change in the consumer code, the consumer can ease the work of the producer and catalyst by only requiring data of a certain freshness.

When the catalyst receives such a wish, it sees if this can be fulfilled with the data it currently has. If the producer does not do lazy commits, this is trivially true. The catalyst knows whether the producer driver has connected to listen for requests for data. If it has not, it can be assumed that the producer driver does not do lazy commits. If the producer on the other hand uses lazy commits and the catalyst is instructed to ensure that it at least has data committed at time t for the set of memory tables M, it must ensure that it has the data or request the producer driver to flush that data. This is the case in Example 7.3.4 where the producer is requested to flush data committed at time 3. If the catalyst already has rows with the time stamp t' where $t \leq t' \neq \infty$ for all $m \in M$, it also has the committed data for t for $m \in M$ due to the flush order rule explained above. It can even be the case that for every $m \in M$, the catalyst has data committed at time $t' > t$. This data can also be used as the

operation is meant to ensure that the catalyst's data is not older than the data that was committed at the given time stamp.

It might, however, be the case that the catalyst has no data committed at or after the wished time stamp t for (some of) the memory tables in M for which accuracy should be ensured. When this happens, the catalyst finds the tables that do not have sufficiently accurate data and requests the producer driver to transfer data for these. It might then be the case that for a memory table m no rows are held in the producer driver's archives in which case the producer driver sends an *empty update* for m, i.e. adds and commits zero rows. For the catalyst, this is still valuable information as the time index can be updated and the accuracy ensured.

Example 7.4.3 (Empty update) *Consider again Example 7.4.2 and assume that no further rows are inserted into X, but that there is a lazy commit at time t_3. If the catalyst sends a request for data committed at time t_3, the producer driver will make an empty update such that the time index maps the time stamp t_3 to the row ID 4: $\tau(t_3) = 4$. Note that we then have $\tau(t_2) = \tau(t_3)$ since no rows were added to X between the commits at t_2 and t_3.*

So if the catalyst is instructed to ensure that it at least has all data committed at time t for the memory tables M, it goes through Algorithm 7.1 where $\mathcal{C}(m,t) = \{\tilde{t} \mid \tilde{t} \in \mathcal{T}(m) \wedge \tilde{t} \geq t\}$ and $\mathcal{T}(m)$ is the set of commit times different from ∞ in the time index for memory table m.

Algorithm 7.1 Find time stamp to consider

Input: A time stamp t and a set of memory tables M

1: **for** $m \in M$ **do**
2: **if** $\mathcal{C}(m,t) = \emptyset$ **then**
3: Request from the producer driver all the unflushed data for m that was committed before or at time t
4: $\Omega_m \leftarrow \{t\}$
5: **else**
6: $\Omega_m \leftarrow \mathcal{C}(m,t) \cup \{t\}$
7: **return** max $\left(\bigcap_{m \in M} \Omega_m\right)$

The return value of Algorithm 7.1 is the newest time stamp for which data can be considered. That means that if the algorithm is invoked for a time stamp t and a set of of memory tables M and returns $\tilde{t}$, it holds that $\tilde{t} \geq t$ and that the catalyst now holds all data that was committed at time $\tilde{t}$ for all $m \in M$. Line 2–3 of the algorithm ensure that the catalyst at least has all the (possibly empty) data sets committed at (or before) time t for each $m \in M$. So we know that data from time t can be considered. But if all $m \in M$ have newer committed data available, the algorithm picks the

maximum time stamp that every m has data for. The found time stamp is returned to the consumer driver which (transparently to the user) ensures that it is used when data is read from the catalyst the next time.

Reads. When a table function reads data, it must also give the catalyst a time stamp that decides what data to include in the result. The time stamp is needed to ensure that data that is too new is not included in the result set as illustrated in the following example.

Example 7.4.4 (Problems in *not* using the time stamp) *Recall the setup for the running example but now assume that both tables X and Y have memory tables. Now consider a scenario where the producer uses lazy commit and the following events take place. (The numbers show how many minutes have passed since the system was started).*

1. *The producer inserts X and Y rows and commits.*
2. *Data for Y is flushed.*
3. *The producer inserts X and Y rows and commits.*
4. *Data for X is flushed.*
5. *A consumer wants 4 minutes accuracy for X and Y.*

The time stamp to use is then for the first commit (4 minutes ago). Note that the last flush for X was 1 minute ago (so all committed data for X is available in the catalyst) while the last flush for Y was 3 minutes ago (so only data committed 4 minutes ago is available in the catalyst). If the catalyst did not use the time stamp and naively returned all data, it would return possibly inconsistent data since X contains data committed 1 minute ago but Y does not.

So by using the time stamp, the catalyst ensures that a consistent snapshot of the committed data is used when returning data to a table function. Based on the time stamp and the time index, the last row to include is found. The last row returned is the row with the biggest row ID that is less than or equal to the minimum of the requested max row ID and the row ID found from the time stamp. Formally, if the minimal requested row ID is i_{min}, the maximal requested row ID is i_{max}, the time stamp is t, and $\hat{\tau}(t)$ is a function giving the time index mapping from the biggest time stamp smaller than or equal to t to a row ID or -1 if this is undefined, then all returned rows have their row IDs in the set

$$\Delta = \{n \mid n \in \mathbb{N}, i_{min} \leq n \leq \min(i_{max}, \hat{\tau}(t))\}$$

Note that the number of returned rows may be different from $|\Delta|$. For a single memory table it is not given that it has all (or even any of) the rows with row IDs in Δ.

If the catalyst has not been instructed to ensure a certain accuracy, the table function will use a special time stamp that says that all committed data must be considered (i.e., the catalyst must hold data committed at or before the current time and the time stamp is set to the current time).

Registering Rows as Being Used A consumer driver can *register* rows with row IDs in a given interval as being used to ensure that they are not deleted from the catalyst while a consumer query should be able to read them there. To register rows as used, corresponds to getting a shared lock. Rows that are registered as being used cannot be deleted from the catalyst. Note that it is not enough to consider rows currently being read as used. A single consumer query may need data from different memory tables or from the same table more than once. In between two reads, the catalyst should not have deleted rows that were within the desired interval of rows in the first read. Therefore, rows should be registered as used before the query starts and *deregistered* after it finishes (the consumer driver does this automatically and transparently as will be explained in Section 7.5).

Only rows that are not already materialized can be registered as used by a consumer. Already materialized rows, can be read from the DW tables and should not block the catalyst from freeing memory. Rows can, on the other hand, be materialized while they are still registered as used. When this happens, the rows will for some time be available both in the DW and in the catalyst. But due to the consumer driver's use of the minmax table, a consumer will only see one instance of each row. This is explained in Section 7.5.

When the producer has performed a materialization, the producer driver informs the catalyst about this. The catalyst uses this to decide which rows it can delete. Rows that are materialized and not registered as being used, can safely be deleted such that the memory can be reused. Deletion is done automatically by the catalyst when more space is needed. Since materialization happens together with commit, it is the case that the rows to materialize have row IDs within a given interval. It is therefore also the case, that the catalyst only has to free one continuous block of memory for each memory table and there is no need to use maps over free regions or similar techniques.

7.5 Consumer Side

In this section, the consumer driver is described. Like the producer driver, the consumer driver is defined by an extension of the JDBC Connection interface. This

extension adds methods for defining how accurate data read from the catalyst has to be. Further, the consumer driver (transparently to the user) ensures that rows are not deleted from the catalyst while they are needed by a consumer query.

From the consumer's point of view, the consumer driver is executing queries with the READ COMMITTED isolation level. To implement this such that it works as expected for both data in the DW and in the catalyst, the driver actually executes queries towards the DW in the REPEATABLE READ isolation level.

Registering Rows as Used Before a query is executed, the consumer driver has to register row IDs as used. As explained in the previous section, this is done to ensure that the rows that exist in the catalyst when the query starts, continue to exist while the query is executed. The row IDs to register as used are those in the range defined by the minmax table, i.e., from the first row that is not materialized when the query starts to the last row that is committed when the query starts. To make sure that rows will not disappear from the catalyst while a query is running, the consumer driver will whenever a method executing a query is invoked, first read values from the minmax table and try to register them with the catalyst. This might fail if a materialization is done between the time the consumer driver reads the values and the time it gives them to the catalyst (recall that the catalyst only allows row IDs of non-materialized rows to be registered as used). In that case, the consumer driver ends the transaction, starts a new transaction and reads values from the minmax table and tries to register them. To avoid starvation problems, the catalyst gives priority to consumers retrying to register values. When the values from the minmax table are registered, the query is executed. After the query is executed, the consumer driver deregisters the values.

Ensuring Accuracy of Read Data The consumer driver also provides the consumer with methods that determine how old data from the catalyst is allowed to be. This is relevant when the producer uses lazy commits. A consumer can then explicitly tell the catalyst how accurate data it needs. If data of the given accuracy or newer data already exist in the catalyst, the producer and the catalyst are released from the burden of flushing data. If the catalyst, on the other hand, does not hold sufficiently fresh data, it requests the producer to flush the needed data. But this happens on-demand and only for the needed data. Note that if these methods are not used, the default is that the catalyst holds all committed data.

Concretely, the JDBC Connection interface is extended with methods `ensureAccuracy(...)` that take a time interval and memory table names as arguments. When these are invoked, the consumer driver passes the wanted accuracy to the catalyst that returns a time stamp for which it has the committed data and that is accurate enough. The value is stored in the DW in a session variable such that it is available for the table function.

Reading Data with the Table Function The consumer driver itself does not read rows from the catalyst. Instead the DW reads rows through a table function, i.e., a stored procedure that returns a set of rows with a structure like rows in a table in the DW. The table function takes as arguments the name of the memory table to read data for and the minimal and maximal row ID of rows to read. When the table function wants to read rows from the catalyst, it also gives the catalyst a time stamp that defines how fresh the data must be (as explained in Section 7.4). Although the row ID arguments can be used to limit the result set in other ways, the normal usage of the minimal row ID is to avoid that the catalyst returns rows that are already materialized when the query begins. This value is defined such that rows with lower row IDs have already been materialized and should be read from the DW. Only from the found value and up, the rows should be read from the catalyst. The normal usage of the maximal row ID is to avoid that the catalyst returns rows that are not committed when the query begins. It is defined to mean that rows with a greater row ID are not committed yet. If this value is read once and reused, it does not affect the query if more rows are committed later.

Example 7.5.1 (Use of the minmax table) *Consider again the state of the minmax table after the materialization in Example 7.3.6 and assume that the producer inserts and commits two rows that get the row IDs 3 and 4, respectively. In the minmax table, the min value is then 3 and the max value is 4.*

A consumer driver now reads these values from the minmax table and successfully registers them. When the table function is given these values, it reads the two new rows from the catalyst. Rows with a row ID less than 3 should not be read since they were already in the DW table when the query started. Now assume that the consumer's query is expensive and involves reading data from the memory table twice. After the first time data is read, but before the second time, the producer inserts and commits some new rows that get row IDs greater than 4. These rows did not exist when the query started. To avoid that the query sees them, the table function is still given the previously read values (i.e., $min = 3$ and $max = 4$).

Finally, assume that while the consumer's query is executing, the producer performs a materialization such that all the new rows also become available in the DW table. The consumer query is still able to read the rows from the catalyst (since they are registered as used and thus cannot be deleted). The consumer query does not get the same rows from the DW table (since it is running in REPEATABLE READ mode and the rows were not in the DW when the query began). So the consumer sees every row that existed when the query began exactly once. The rows that were committed after the query began are not seen.

Transparency To make these things transparent to the end user, a view over a DW table and its associated memory table can be defined. If the view definition uses

the minmax table directly, we find the same rows in the view every time the view is used within one query (recall that the consumer connection is put in REPEATABLE READ mode). So for each DW table for which a memory table also exists, a view should be defined as

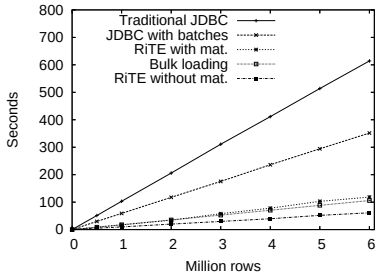
```
CREATE VIEW v AS SELECT * FROM dwtable UNION ALL
SELECT * FROM tablefunction('dwtable', (SELECT
  min FROM minmax), (SELECT max FROM minmax))
```

If the view v is used instead of $dwtable$ in queries, the end user does not have to think about if rows are read from the tables in the DW or from the catalyst. Since the consumer driver behind the scenes is using the REAPEATABLE READ isolation level, a single query that uses the view many times sees the same values from the minmax table and thus the same set of rows in the view. But the consumer driver starts a new transaction for each query and some rows might have been updated when a query is re-executed. In other words, non-repeatable reads are possible such that the isolation level in effect is READ COMMITTED as promised by the driver.

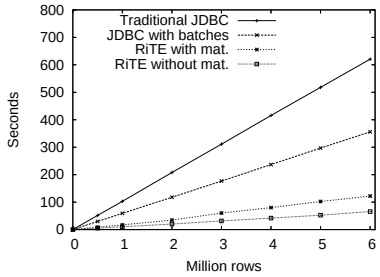
7.6 Performance Study

Setup We now present a performance study of the RiTE prototype. The prototype (www.cs.aau.dk/∼chr/RiTE) consists of 1) Java JDBC database drivers for producers and consumers, 2) the catalyst (Java), and 3) a C implementation of a PostgreSQL table function. The prototype shows a working solution for a DW based on PostgreSQL [94] version 8.1 running on a Linux x86 platform. However, the applied principles are general and could be used for most DBMSs. The catalyst is completely DBMS-independent while the JDBC drivers have few (marked) PostgreSQL dependencies. The table function is, of course, highly dependent on the hosting DBMS platform. The experiments have been carried out on a 3GHz Pentium 4 PC with 3.2GB RAM and four SATA disks of which one is used for DW data, one for PostgreSQL's write-ahead logs, one for source data and one for binary executables and swap area. The PC is running Ubuntu Linux 6.10, Java 6SE, and PostgreSQL 8.1.4. The PostgreSQL configuration can be found at www.cs.aau.dk/∼chr/RiTE. We simulate a producer filling a fact table. The source data originates from TPC-H [110], with the schema modified to a star schema. Rows are inserted into the typical fact table *lineitem* with 6 integer columns (*custkey*, *datekey*, *orderkey*, *partkey*, *suppkey*, and *quantity*).

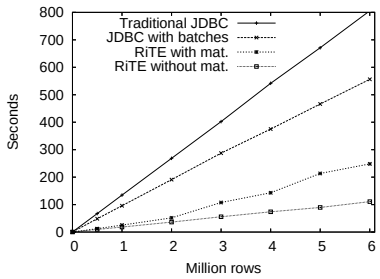
Long Transactions We first consider the performance when inserting many rows into one table with insert statements. We consider a producer application, both when using RiTE and the traditional JDBC driver, and compare this to applications that load



(a) Commit after the last row



(b) Commit pr. 10,000 rows



(c) Concurrent producer and consumer

Figure 7.2: Performance results

the same data set by doing multirow inserts with JDBC *batches* and bulk loading, respectively. Prepared statements are used where applicable. The values to insert are read from a text file. The producer runs in one long transaction and commits after the last insert. The same producer application is used throughout, with only the lines setting up the DW JDBC connection and doing the final commit changed. A suitably modified JDBC application is used to test JDBC batches with a batch size of 10,000 rows. Bulk loading is done by letting a modified application write the data to a comma separated file and then let the PostgreSQL server read the file directly.

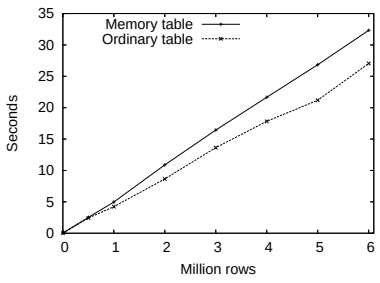
The graph shown in Figure 7.6(a) shows the results, which are 9,646 rows/second (traditional JDBC driver), 17,088 rows/second (JDBC batches), 49,878 rows/second (RiTE with materialization), 56,846 rows/second (bulk loading), and 98,723 rows/second (RiTE without materialization). As the systems scales linearly, the speeds are based on the line slopes. The best throughput is obtained when using RiTE without materialization. The throughput is then 74% higher than for bulk loading.

Short Transactions The experiment is now repeated, but with commits for every 10,000 rows. As bulk loads do not commit during the load, they are not used. The results plotted in Figure 7.6(b) show that JDBC's performance is not affected. For

JDBC batches, the throughput drops slightly (to 16,841 rows/second). With RiTE, the producer can now insert 47,686 rows/second with materialization and 90,356 rows/second without materialization. Similar results are obtained for commits for every 100,000 rows.

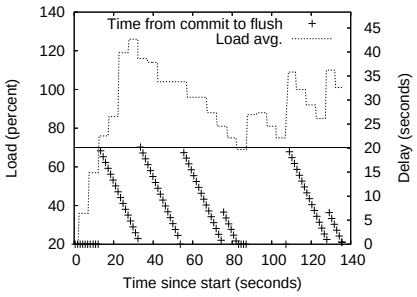
Influence from a Consumer The 10,000 row commit experiment is repeated, but this time a consumer application simultaneously performs the query `SELECT SUM (quantity)` (reading all rows) on the *lineitem* table (which has a memory table when using RiTE). The query is re-executed right after returning its results, so the system is constantly loaded. The results plotted in Figure 7.6(c) show that the JDBC application can insert 7,451 rows/second whereas JDBC with batches can insert 10,862 rows/second. For RiTE, the producer can insert 22,437 rows/second with materialization and 54,111 rows/second without materialization. Thus, performance is affected, but the relative advantage of RiTE remains.

Read Performance We now compare how fast data can be read from a DW table and a memory table. The data sets used in the previous experiments are loaded once (into a memory table or a DW table, depending on what is being tested). Then all rows are read 6 times from PostgreSQL's terminal and the time usages measured. The first recorded time usage is not considered as this is used to let PostgreSQL buffer the data to make fair comparisons. The performance results are plotted in the shown graph. From the slopes of the lines, we estimate that the system reads 219,168 rows/second from a non-memory (but buffered) table whereas it reads 182,786 rows/second from a memory table. The difference is due to that when data is read from a memory table, type conversions from Java types to the host machine's native types are performed and data is transferred from the catalyst to the DBMS. There is thus a small overhead for RiTE reads.



Lazy Commit Delays We now consider a producer that constantly inserts rows and commits once per second. The producer uses lazy commit and its flush policy is to flush when the system load is below 70% or 20 seconds have passed since the last flush. While the producer runs, a load simulator generates randomness in the CPU load. In the shown graph, the dotted line shows the CPU load (to be read relatively to the left Y axis) at different times while a cross at (x, y) shows that data committed at time x waits y seconds before it is flushed (where y should be read relatively to the right Y axis). The solid horizontal line shows where 70%

is on the left Y axis and where 20 seconds is on the right Y axis, i.e. it shows the "limits" for the policy. It is seen that at first, the CPU load is below 70% and data is flushed with no delay after a commit. After appx. 12 seconds, the load gets higher than 70% and there are up to 20 second delays between commit and flush. When a flush is done, all committed data is flushed (notice how the crosses lie on lines with a negative slope). After appx. 82 seconds, the CPU load gets below 70% and data is flushed before 20 seconds have passed since the last flush (see the short line of crosses around 80 seconds). Also when the producer terminates, it flushes all data, so the delay is below 20 seconds.



Summary From the experiments it is clear that RiTE provides a significant performance increase: between 4 and 10 times for inserts and 2 to 6 times for inserts with concurrent reads.

7.7 Related Work

The issue of moving data from one place to another has a long tradition in both research and industry. The ETL process may be implemented in a materialized or virtual way. Linking external data sources into a target system is discussed in the context of federations. Using wrapper-like technologies [97], DW systems gain access to the underlying data. Selection and transformation routines are directly applied to the external data; the result directly goes into the DW tables. Materialization implies the physical movement of data into the target system. Techniques are ranging from import of flat files to (a)synchronous replication [83]. While replication may conceptually provide functionality somewhat similar to RiTE, current replication techniques are (unlike RiTE) limited to simple transformations and certain (cooperative) source systems, and put additional overhead on the data sources. In comparison, RiTE takes advantage of the special characteristics of right-time DWs, and can thus provide quickly-available data at bulk-load insert speeds. This can be provided for any type of source system and any type of transformation, as these parts are handled by the ETL code. With RiTE, the producer decides when to make data available to all consumers and when to move data around (by using the commit materialize operations, respectively).

From a conceptual point of view, incorporating external data into a single DW database requires a consistent global view. Starting with database snapshots [1], sig-

nificant research was devoted to that problem in recent years under the notion of materialized views [47]. Initial work like [26, 126] investigated methods to establish a consistent view over multiple sources or updating multiple views with data coming from a single source [41]. All these mechanism are orthogonal to RiTE and may be applied on top of our middleware. More closely related is research documented in [98] rolling global DW states forward to certain points in time. However, this approach requires an explicit trigger while our approach is fully demand-driven. A similar approach with implicit instructions based on the notion of policies is outlined in [46]; in contrast, we focus on the efficient implementation (catalyst) in combination with a transactionally consistent view on the data source and thus go much further.

The state of the art of continuous loading is summarized in [63]. Compared to that, RiTE gives the producer full control over the units of work to commit together and is flexible with respect to persistency guarantees versus load speed. Further, RiTE is more flexible with respect to freshness of data and offers lazy commit which can make data available in the DW on demand.

The MySQL [71] DBMS offers a memory storage engine for fast, but non-persistent, storage and access. Unlike MySQL, RiTE has functionality for migrating rows from memory to the database (i.e., materialization). The MySQL main memory storage engine also obviously does not scale to DW data volumes. Additionally, RiTE allows rows to be added by while other rows are read, whereas MySQL uses table locking when rows are inserted into a memory table. MySQL also offers INSERT DELAYED syntax where many inserts can be bundled and written in one block when the target table is not in use. This holds back INSERT data similarly to RiTE, but in RiTE the producer controls when to make the rows available (at commit time). INSERT DELAYED is slower than normal INSERT if the target is not in use and should be used carefully. In contrast, RiTE provides a speed-up for the producer also when no consumers exist.

7.8 Conclusion and Future Work

Motivated by the need for a solution that makes inserted data available quickly, while still providing bulk-load insert speeds, this chapter presented the middleware RiTE ("Right-Time ETL"). A data producer (ETL) can insert data that becomes available to data consumers on demand. To make this possible, RiTE introduces an innovative main-memory based catalyst and supports a number of policies that control the bulk movement of data based on user requirements for persistency, availability, freshness, etc. RiTE works completely transparently to both producer and consumers. A prototype has been integrated with an open-source DBMS, and experiments have shown

that RiTE provides “the best of both worlds”, i.e., INSERT-like data availability, but with bulk-load speeds (up to 10 times faster).

There are many interesting directions for future work. Logging could be added to the catalyst such that persistency guarantees can also be given when materialization is not done. Possibilities for letting rules provide transparent updating and deletion of rows inserted into memory tables would also be relevant. Fast inserts could then be performed on the fly and a data cleansing procedure could correct mistakes or delete bad rows before materialization. The catalyst could also be implemented as a module in the underlying DBMS since an even better performance could be obtained when no repetitive type conversions from Java types to the DBMS’ native types would have to take place then. A related task is to allow indexes and constraints to be declared on memory tables.

Chapter 8

Summary of Conclusions and Future Research Directions

This chapter summarizes the conclusions and directions for future work presented in Chapters 2–7 and Appendix A.

8.1 Summary of Results

This thesis is about aspects of specification and development of data warehouse technologies for complex web data. The work that led to this thesis was primarily done in relation to the European Internet Accessibility Observatory (EIAO) project for which a DW for accessibility data and supporting DW technologies were specified and developed. The thesis has thus among other things presented how (accessibility) data about web resources can be modeled in a DW, how to handle OWL data efficiently, and how to do flexible import and export of relational data via XML. As the source data and the way it is fetched may change frequently on the Web, regression test of ETL software is very relevant in a web setting and the thesis also presented a framework that makes regression test of ETL software easy to start. The thesis also proposed a solution for how to make data available in a DW on demand while preserving the speeds from bulk loading at regular intervals. In a web setting where data constantly becomes available from online resources, this makes it possible for DW users access the newest data immediately instead of having to wait for a load to occur. The developed technologies were all made in a general general way and although the work was done in relation to the EIAO project, the developed technologies can thus also be applied in other environments. In the following, we go through each of the presented chapters and summarize the most important results.

Chapter 2 surveyed the possibilities for using open source BI products as of End 2004. Considering the fact that use of open source BI tools in industry is not common

while, e.g., open source web servers are used extensively in industry, it was interesting to investigate the possibilities for using open source BI products. Further, only open source products were to be used in the EIAO project. The chapter presented some of the commonly used open-source licenses. Then three Extract-Transform-Load (ETL) ETL tools, three On-Line Analytical Processing (OLAP) servers, two OLAP clients, and four Database Management Systems (DBMSs) were considered and evaluated against criteria relevant to the use of BI in industry. The chapter concluded that the DBMSs are the most mature of the tools and applicable to real-world projects. On the other hand, it was concluded that the ETL tools were not mature and in general not ready for use in industry.

Chapter 3 presented EIAO DW release 1 (from Mid 2006) which is a general and scalable web warehouse built to make analysis of complex data about (in)accessibility of web resources easy, fast, and reliable. This included to calculate complex aggregation results giving a number describing the accessibility of web resources. EIAO DW is believed to be the first general and scalable DW for accessibility data. The chapter gave a brief introduction to the field of accessibility and to the entire architecture used in the EIAO project. Then the conceptual, logical, and physical models were presented. These are made such that it is easy to add new accessibility test types since no schema changes are required for this. The ETL procedure for EIAO DW extracts data to insert into the DW from RDF source data and was also presented. Bad performance when extracting the RDF based source data is, however, a problem for the used solution.

Chapter 4 presented 3XL, a proposal for how to store very large Web Ontology Language (OWL) graphs efficiently by making a specialized database schema for the data to store. This was motivated by the previous experiences with performance problems with large RDF data sets in general schemas. 3XL focuses on the subset of RDF graphs that are also OWL Lite graphs since they have some convenient characteristics that are used for the schema generation. The chapter presented how each of the supported OWL constructs is reflected in the specialized schema. In contrast to a generic schema with few large and narrow tables, 3XL has many and wide tables. Further, the chapter presented how addition of data is handled in 3XL as well as how results for queries in the form of triples are found by means of SQL queries. The chapter also presented a theoretical analysis that showed that 3XL inserts much fewer rows than a solution using a generic schema. This results in less storage overhead from rows. In a presented example with data about 10^7 instances, a specialized schema created by 3XL required 2.89GB storage (or 10.34GB if so-called multiproperty tables were used) whereas a generic schema required 11.79GB storage.

Chapter 5 investigated automatic and effective bidirectional transfer between relational and XML data. This was motivated by the increasing exchange of relational data through XML based technologies such as web services. To set such exchange

up manually is cumbersome and a lot of hand-coding is needed. As a remedy to this situation, the chapter presented RELAXML where the user must only specify the structure of the XML and what data to exchange. The chapter presented which conditions must be fulfilled to make it possible to insert the exported data into a new database or to be able to detect changes made to the XML (by only considering the original database and the XML document) and update the database to reflect these changes. Further, the chapter presented technical solutions for detecting problems with the data (e.g., inconsistent updates and dead links referencing missing data) and algorithms used in the implementation of RELAXML. A performance study showed that the solution has a reasonable overhead compared to specialized, hand-coded solutions.

Chapter 6 considered regression test of ETL software. ETL software tends to be complex and error prone and may often be changed to increase performance or to handle changed data sources. Regression test is thus very useful for ETL software but traditionally it has required large manual efforts to set up. The chapter pointed out crucial differences between testing in "normal" software development and ETL development and, based on these, the tool ETLDiff was presented. The chapter presented how ETLDiff analyzes the DW schema and detects which parts of the data should not change between ETL runs on the same source data. Based on the analysis and optional user specification about what data to consider, ETLDiff compares test results to previous test results or other reference results and points out differences. When ETLDiff is used, a regression test can be set up in minutes instead of in days as when manual coding is done. The chapter also presented a performance study of a prototype of ETLDiff. The results showed that the running time scales linearly in the data size and that the solution is efficient enough to be used for regression testing on a desktop PC.

Chapter 7 investigated how to insert data into so-called right-time DWs. Traditionally, data has been bulk loaded into DWs at regular intervals but recently it has become popular to insert new data as soon as it appears by using traditional SQL INSERT statements. This makes data available quickly, but performance suffers when the data amounts grow as when, e.g, click-streams are considered. There is thus a need to be able to make data available quickly while still preserving a high insert performance. The chapter presented the middleware system RiTE that provides such a solution which works transparently to both consumers and the producer. When RiTE is used, data can be inserted quickly by a producer and become available to consumers exactly when needed. The chapter presented how RiTE does this by using a local buffer at the producer side which is flushed when needed and by introducing a novel main-memory based catalyst that it is fast to insert data into and which can be accessed transparently to the user from the DW. Further, the chapter presented how movement of data between the local buffer, the catalyst, and the DW is supported

in many different ways and how it is ensured that the client sees consistent data (i.e., how transactions are supported). The chapter also presented experiments that showed that a prototype of RiTE provides INSERT-like data availability, but up to 10 times faster, i.e., with bulk-load speeds.

Appendix A presented the conceptual model for release 2.0 (from Mid 2007) of the EIAO DW. Compared to the conceptual model presented in Chapter 3, several changes have occurred to reflect the changed requirements and available data from the entire EIAO project.

The thesis has thus presented aspects of data warehouse technologies for complex web data. The work has primarily been done in relation to the EIAO project for which EIAO DW has been developed and only open source software has been used. But the developed technologies are general and can also be applied in other DW projects.

8.2 Research Directions

Several directions for future work remain for the work presented in this thesis. To monitor the development and progress for open source BI software seems to be as relevant as ever. New products and projects are launched and existing products extended. And although the price for buying commercial solutions may not be a problem for big enterprises, it matters for small companies. But with open source products, the path to start using BI may become easier (i.e., cheaper) to follow. So in the future, complete open source BI solutions should be made available to organizations starting to use BI.

The development of EIAO DW described in Chapter 3 continues for the upcoming 2.1 and 2.2 releases. In these releases, the Observatory (and thus also the DW) will among other things have support for results for PDF files and JavaScript. Release 2.2 will be put into large-scale production and deliver monthly access to accessibility evaluations of 10,000 European web sites.

Several interesting directions exist for the 3XL system presented in Chapter 4. First of all it should be implemented and used such that practical experiences can be gained. Further, the design could be extended to support more or all of the OWL Lite constructs. It would also be very useful to add support for a query language in a way that exploits the specialized schema.

For RELAXML described in Chapter 5 there are also interesting directions for future work. It seems attractive to investigate how the concepts that define what data to export/import and the structure definitions that define the structure of the XML can be made even more flexible. For example, it could be considered to introduce parameterized concepts that can then be used as a single element in another concept. Further, the possibilities for generating and compiling specialized code on the fly for a given concept and structure definition should be investigated. By doing that it may

be possible to reduce or even get rid of the overhead from using a general tool as RELAXML compared to a hand-coded specialized solution.

Also with respect to testing of ETL software there are interesting directions for continuations of the work in Chapter 6. In general it is interesting to move agile methods (including continuous testing) into the BI field and the possibilities for this should be investigated. With respect to ETLDiff, it would be interesting to extend the tool to support other kinds of tests such as audit tests where the source data set and loaded data set are compared.

As the acceptable delays before insertion of new data into the DW get smaller and smaller and data sizes bigger and bigger, there is an increased need for a tool like RiTE presented in Chapter 7. This makes it interesting to make it perform even better, for example by implementing it as a specialized module for the host DBMS instead of as a general Java module as now. It could also be exciting to provide traditional DBMS features such as possibilities for updates and deletes of the data *before* it reaches the DW as well as provide indexes etc. Also logging seems very attractive. Now RiTE does not give persistency guarantees before the data has been *materialized* to the underlying DW. But with RiTE support for logging, it would be possible to provide such guarantees while still having an extremely good performance.

In the future more projects using data warehouses with web data or web metadata will appear. It would be beneficial to have a common way to model web resources in DWs such that data from different sources easily could be compared and shared. It also seems very attractive to be able to integrate dynamic data from the Web into a DW by having virtual dimensions or fact tables over resources from the Web such as RDF documents, XML documents, web services, etc. To make DW data available in web format “views” such as RDF is also an interesting idea. In this way, the Semantic Web vision may use the huge amounts of knowledge available in DWs and do reasoning and automatically find new knowledge.

Bibliography

[1] M. E. Adiba and B. G. Lindsay. Database Snapshots. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, pp. 86–91, 1980.

[2] S. Alexaki, V. Chrisophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing Volumiunous RDF Description Bases. In *Proceedings of the Second International Workshop on the Semantic Web*, 2001.

[3] S. Alexaki, V. Chrisophides, G. Karvounarakis, and D. Plexousakis. On Storing Voluminous RDF Descriptions: The case of Web Portal Catalogs. In *Proceedings of Proceedings of the Fourth International Workshop on the Web and Databases*, pp. 43–48, 2001.

[4] N. Alexander, X. Lopez, S. Ravada, S. Stephens, and J. Wang. RDF Data Model in Oracle. Available from `download-uk.oracle.com/otndocs/tech/semantic_web/pdf/w3d_rdf_data_model.pdf` as of 2007-10-10.

[5] G. Antoniou and F. van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.

[6] Apache SW License v1.1. `www.apache.org/licenses/LICENSE-1.1` as of 2007-10-10.

[7] Apache SW License, v2.0. `www.apache.org/licenses/LICENSE-2.0` as of 2007-10-10.

[8] Bee. `sourceforge.net/projects/bee/` as of 2005-05-30[1].

[9] Bee Project. `bee.insightstrategy.cz/en/` as of 2005-05-30.

[1]This bibliography has been created by merging the bibliographies from the individual papers that appear in this thesis. The dates given in "as of ..." for web resources have in general been updated, but for web resources that do not exist anymore or have had their content significantly updated, the date given in the original paper has been kept.

[10] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[11] S. S. Bhowmick, W. K. Ng, and S. Madria. Web Schemas in WHOWEDA. In *Proceedings of the 3rd ACM International Workshop on Data Warehousing and OLAP*, 2000.

[12] P. Bohannon, J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. LegoDB: Customizing Relational Storage for XML Documents. In *Proceedings of 28th International Conference on Very Large Data Bases*, pp. 1091–1094, 2002.

[13] R. Bourret. XML Database Products: Middleware. `www.rpbourret.com/xml/ProdsMiddleware.htm` as of 2007-10-10.

[14] R. Bourret. XML-DBMS. `www.rpbourret.com/xmldbms/index.htm` as of 2007-10-10.

[15] R. Bourret. XML Database Products. `www.rpbourret.com/xml/XMLDatabaseProds.htm` as of 2007-10-10.

[16] R. Bourret, C. Bornövd, and A. Buchman. A Generic Load/Extract Utility for Data Transfer between XML Documents and Relational Databases. In *Proccedings of the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Webbased Information Systems*, pp. 134–143, 2000.

[17] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From XML View Updates to Relational View Updates: old solutions to a new problem. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pp. 276–287, 2004.

[18] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the First International Semantic Web Conference*, pp. 54–68, 2002.

[19] BSD license. `opensource.org/licenses/bsd-license.php` as of 2007-10-10.

[20] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *Proceedings of 26th International Conference on Very Large Data Bases*, pp. 646–648, 2000.

[21] A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison-Wesley, 2003.

[22] D. Chays, S. Dan, P. Frankl, F. I. Vokolos, and E. J. Weyuker. A Framework for Testing Database Applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 147–157, 2000.

[23] C. A. Christensen, S. Gundersborg, K. de Linde, and K. Torp. *A Unit-Test Framework for Database Applications*. TR, Aalborg University, 2006. Available from `www.cs.aau.dk/DBTR` as of 2007-10-10.

[24] CloverETL. `cloveretl.berlios.de` as of 2005-05-30.

[25] G. Cobéna, T. Abdessalem, and Y. Hinnach. *A comparative study for XML change detection*. TR, 2002. Available from `ftp://ftp.inria.fr/INRIA/Projects/verso/VersoReport-221.pdf` as of 2007-10-10.

[26] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting Multiple View Maintenance Policies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 405–416, 1997.

[27] CPL v1.0. `opensource.org/licenses/cpl1.0.php` as of 2007-10-10.

[28] cplusql ETL tool. `sourceforge.net/projects/cplusql/` as of 2007-10-10.

[29] B. Daou, R. A. Haraty, and N. Mansour. Regression Testing of Database Applications. In *Proceedings of the Symposium on Applied Computing*, pp. 285–290, 2001.

[30] C. J. Date. *An Introduction to Database Systems*, 7th Edition. Addison-Wesley, 2000.

[31] U. Dayal and P. A. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM Transactions on Database Systems* 8(2):381–418, 1982.

[32] DbUnit. `dbunit.sourceforge.net` as of 2006-06-09.

[33] H. Engström, S. Chakravarthy, and B. Lings. A Heuristic for Refresh Policy Selection in Heterogeneous Environments. In *Proceedings of the 19th International Conference on Data Engineering*, pp. 674–676, 2003.

[34] Enhydra Octopus. `octopus.objectweb.org/` as of 2005-05-30.

[35] European Commission. Information Society – eInclusion & eAccessibility. europa.eu.int/information_society/policy/accessibility/index_en.htm as of 2006-08-28.

[36] European Commission. Information Society – eInclusion & eAccessibility. europa.eu.int/information_society/policy/accessibility/z-techserv-web/index_en.htm as of 2006-08-28.

[37] European Internet Accessibility Observatory. eiao.net as of 2007-10-10.

[38] Eurostat. Nomenclature of territorial units for statistics - NUTS Statistical Regions of Europe. ec.europa.eu/comm/eurostat/ramon/nuts/home_regions_en.html as of 2007-10-10.

[39] EU Web Accessibility Benchmarking Cluster. Unified Web Evaluation Methodology version 0.5. www.wabcluster.org/uwem05/ as of 2007-10-10.

[40] M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.

[41] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, and L. Sheng. Optimizing Refresh of a Set of Materialized Views. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pp. 1043–1054, 2005.

[42] J. Gardner. Materialized Views in PostgreSQL. jonathangardner.net/PostgreSQL/materialized_views/matviews.html as of 2007-10-10.

[43] GNU GPL License. www.gnu.org/copyleft/gpl.html as of 2005-05-30.

[44] GNU LGPL License. www.gnu.org/copyleft/lgpl.html as of 2005-05-30.

[45] gnuOLAP. gnuolap.sourceforge.net/ as of 2007-10-10.

[46] H. Guo, P.-Å. Larson, and R. Ramakrishnan. Caching with 'Good Enough' Currency, Consistency, and Completeness. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pp. 457–468, 2005.

[47] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[48] S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In *Proceedings of the First International Workshop on Practical and Scalable Semantic Systems*, 2003.

[49] Intelligent Systems Research. JDBC2XML: Merging JDBC Data into XML Documents. `www.intsysr.com/jdbc2xml.htm` as of 2007-10-10.

[50] International Organization for Standardization/International Electrotechnical Commission. *XML-Related Specifications (SQL/XML)*. INCITS/ISO/IEC 9075-14:2003. 2003.

[51] Java OLAP Interface (JOLAP). `www.jcp.org/en/jsr/detail?id=69` as of 2007-10-10.

[52] M. R. Jensen, T. Holmgren, and T. B. Pedersen. Discovering Multidimensional Structure in Relational Data. In *Proceedings of the 6th International Conference on Data Warehousing and Knowledge Discovery*, pp. 138–148, 2004.

[53] JFreeChart. `www.jfree.org/jfreechart` as of 2007-10-10.

[54] JPivot. `jpivot.sourceforge.net/` as of 2005-05-30.

[55] JUnit. `junit.org` as of 2007-10-10.

[56] R. Kimball and R. Merz. *The Data Webhouse Toolkit*. Wiley, 2000.

[57] R. Kimball, L. Reeves, M. Ross, and W. Thornthwaite. *The Data Warehouse Lifecycle Toolkit*. Wiley, 1998.

[58] R. Kimball and M. Ross. *The Data Warehouse Toolkit*, 2nd Edition. Wiley, 2002.

[59] A. Kiryakov, D. Ognyanov, D. Manov. OWLIM – a Pragmatic Semantic Repository for OWL. In *Proceedings of the 2005 International Workshop on Scalable Semantic Web Knowledge Base Systems*, pp. 182–192, 2005.

[60] S. U. Knudsen, T. B. Pedersen, C. Thomsen, and K. Torp. RELAXML: Bidirectional Transfer between Relational and XML Data. In *Proceedings of the 9th International Database Engineering & Application Symposium*, pp. 151–162, 2005.

[61] S. U. Knudsen and C. Thomsen. RELAXML*: A Tool for Transferring Data Between Relational Databases and XML Files*. Master thesis, Aalborg University, 2004.

[62] Lemur OLAP library. `www.nongnu.org/lemur/` as of 2007-10-10.

[63] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Waltzke. Transaction Reordering and Grouping for Continuous Data Loading. In *Proceedings of the First International Workshop on Business Intelligence for the Real-Time Enterprises*, pp. 34–49, 2006.

[64] MaxDB. `www.mysql.com/products/maxdb/` as of 2005-05-30.

[65] Microsoft Corporation. SQL Server Integration Services. `www.microsoft.com/sql/technologies/integration/default.mspx` as of 2007-10-10.

[66] Y. Mohamad, D. Stegemann, J. Koch, and C. A. Velasco. imergo: Supporting Accessibility and Web Standards to Meet the Needs of the Industry via Process-Oriented Software Tools. In *Proceedings of Computer Helping People with Special Needs: 9th International Conference*, 2004.

[67] Mondrian components. `perforce.eigenbase.org:8080/open/mondrian/doc/index.html`, as of 2007-10-10.

[68] Mondrian FAQs. `perforce.eigenbase.org:8080/open/mondrian/doc/faq.html` as of 2007-10-10.

[69] MonetDB. `monetdb.cwi.nl/` as of 2007-10-10.

[70] Mozilla Public License v1.1. `www.mozilla.org/MPL/MPL-1.1.html` as of 2007-10-10.

[71] MySQL. `mysql.com` as of 2007-10-10.

[72] MySQL Case Studies. `mysql.com/it-resources/case-studies/` as of 2005-05-30.

[73] MySQL Database Server. `mysql.com/products/mysql/` as of 2005-05-30.

[74] Netbryx Technologies. DataDesk v. 1.0. `www.netbryx.com/DataDesk.aspx` as of 2005-02-21.

[75] A. Nietzio. *EIAO Deliverable 3.2.1: 2nd Version of ROBACC WAMs*. 2006. Available from `eiao.net/publications/` as of 2007-10-10.

[76] The OLAP Report Market share analysis. `www.olapreport.com/market.htm` as of 2005-05-30.

[77] OpenETL. `openetl.tigris.org` as of 2007-10-10.

[78] Object Management Group. UML Resource Page. `omg.org/UML/` as of 2007-10-10.

[79] Ontotext Lab, Sirma Group Corp. BigOWLIM Semantic Repository, 2006. Available from `www.ontotext.com/owlim/big/BigOWLIMSysDoc.pdf` as of 2007-10-10.

[80] Open Source ETL (OpnSrcETL). `opnsrcetl.sourceforge.net/` as of 2005-05-30.

[81] OpenROLAP. `openrolap.sourceforge.net` as of 2005-05-30.

[82] A. Owens. *Semantic Storage: Overview and Assessment*. TR, University of Southampton, 2005.

[83] T.M. Özsu and P. Valduriez. *Principles of Distributed Database Systems*, 2nd Edition. Prentice Hall, 1999.

[84] Z. Pan and J. Heflin. *DLDB: Extending Relational Databases to Support Semantic Web Queries*. TR, Leigh University, 2004.

[85] T. B. Pedersen. How Is BI Used in Industry?: Report from a Knowledge Exchange Network. In *Proceedings of the 6th International Conference on Data Warehousing and Knowledge Discovery*, pp. 179–188, 2004.

[86] T. B. Pedersen and C. S. Jensen. Multidimensional Database Technology. *IEEE Computer*, 34(12):40–46, 2001.

[87] T. B. Pedersen and C. Thomsen. *EIAO Deliverable 6.1.1.1-2: Functional specification and architecture of EIAO DW, R2.0*. 2007. Available from `eiao.net/publications/` as of 2007-10-10.

[88] T. B. Pedersen and C. Thomsen. *EIAO Deliverable 6.1.1.1-2: Functional specification and architecture of EIAO DW, R2.0, Appendix A*. 2007. Available from `eiao.net/publications/` as of 2007-10-10.

[89] T. B. Pedersen and C. Thomsen. *EIAO Deliverable 6.3.3.1: Documentation for functional EIAO DW, R1.0bis*. 2006.

[90] J. M. Pérez, R. Berlanga, M. J. Aramburu, and T. B. Pedersen. *Integrating DWs with the Web Using XML: A Survey*. TR, 2006. Available from `www.cs.aau.dk/DBTR/` as of 2007-10-10.

[91] O. Perlick, A. Nietzio, H. Heck, D. Clemens, and P. Bertini. *EIAO Deliverable 3.1.1: 1st Version of ROBACC WAMs*. 2006. Available from `eiao.net/publications/` as of 2007-10-10.

[92] L. Peters. Change Detection in XML Trees: a Survey. In *Proceedings of the 3rd Twente Student Conference on IT*, 2005. Available from `referaat.ewi.utwente.nl/documents/2005_03_B-DATA_AND_APPLICATION_INTEGRATION/` as of 2007-10-10.

[93] pocOLAP - the "little" OLAP-project. `pocolap.sourceforge.net/` as of 2007-10-10.

[94] PostgreSQL. `postgresql.org` as of 2007-10-10.

[95] The R Project for Statistical Computing. `www.r-project.org` as of 2007-10-10.

[96] G. Reese. *Database Programming with JDBC and Java*. O'Reilly, 2000.

[97] M. T. Roth and P. M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pp. 266–275, 1997.

[98] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 129–140, 2000.

[99] SAX. `www.saxproject.org` as of 2007-10-10.

[100] A. Schmetzke. Web Accessibility Survey Site. `library.uwsp.edu/aschmetz/Accessible/websurveys.htm` as of 2006-08-28.

[101] Self-Organizing Databases. `research.csc.ncsu.edu/selftune/` as of 2007-10-10.

[102] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases*, pp. 302–314, 1999.

[103] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. E. Funderburk. Querying XML Views of Relational Data. In *Proceedings of 27th International Conference on Very Large Data Bases*, pp. 261–270, 2001.

[104] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Databases as XML Documents. In *Proceedings of 26th International Conference on Very Large Data Bases*, pp. 65–76, 2000.

[105] M. H. Snaprud, C. S. Jensen, N. Ulltveit-Moe, J. P. Nytun, M. E. Rafoshei-Klev, A. Sawicka, and Ø. Hanssen. Towards a Web Accessibility Monitor. In *Proceedings of the Second European Medical and Biological Engineering Conference*, 2002.

[106] G. Spofford and E. Thomsen. *MDX Solutions: With Microsoft SQL Server Analysis Services*, Wiley, 2001.

[107] M. Stonebraker and L. Rowe. *The Postgres papers*. TR, UC Berkeley, 1987.

[108] Sun Microsystems. Java SE Technologies - Database. `java.sun.com/javase/technologies/database` as of 2007-10-10.

[109] C. Thomsen and T. B. Pedersen. A Survey of Open Source Tools for Business Intelligence. In *Proceedings of 7th International Conference on Data Warehousing and Knowledge Discovery*, pp. 74–84, 2005.

[110] TPC-H. `tpc.org/tpch/` as of 2007-10-10.

[111] N. Ulltveit-Moe, M. G. Olsen, C. Thomsen, A. B. Pillai, T. Gjøsæter, and T. B. Pedersen. *EIAO Deliverable 5.1.1.1-2: Functional Specification*. 2006. Available from `eiao.net/publications/` as of 2007-10-10.

[112] J. Vacca. *VRML: Bringing Virtual Reality to the Internet*. Academic Press, 1997.

[113] Watchfire Corp. Watchfire WebXACT. `webxact.watchfire.com` as of 2007-10-10.

[114] Web Server Survey Achives. `news.netcraft.com/archives/web_server_survey.html` as of 2005-05-30.

[115] K. Wilkinson, C. Sayers, H. Kuno, D Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proceedings of the first International Workshop on Semantic Web and Databases*, pp. 131–150, 2003.

[116] D. Willmor and S. Embury. A Safe Regression Test Selection Technique for Database-Driven Applications. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 421–430, 2005.

[117] World Wide Web Consortium. OWL Web Ontology Language Reference. 2004. Available from `w3.org/TR/owl-ref/` as of 2007-10-10.

[118] World Wide Web Consortium. RDF Vocabulary Description Language 1.0: RDF Schema. 2004. Available from `w3.org/TR/rdf-schema/` as of 2007-10-10.

[119] World Wide Web Consortium. Resource Description Framework (RDF). `w3.org/RDF/` as of 2007-10-10.

[120] World Wide Web Consortium. Evaluation and Reporting Language (EARL) 1.0 Schema. Working draft. Available from `w3.org/TR/EARL10-Schema/` as of 2007-10-10.

[121] World Wide Web Consortium. Web Accessibility Initiative. `w3.org/WAI/` as of 2007-10-10.

[122] World Wide Web Consortium. Web Content Accessibility Guidelines 1.0. 1999. Available from `w3.org/TR/WCAG10/` as of 2007-10-10.

[123] World Wide Web Consortium. Web Content Accessibility Guidelines 2.0. Working draft. Available from `w3.org/TR/WCAG20/` as of 2007-10-10.

[124] XML for analysis. `xmla.org/download.asp?id=2` as of 2005-05-30.

[125] X. Zeng. *Evaluation and Enhancement of Web Content Accessibility for Persons With Disabilities*. PhD thesis, University of Pittsburgh, 2004.

[126] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, pp. 146–157, 1996.

Appendix A

Conceptual model for EIAO DW Release 2

In this appendix, the conceptual model for the data warehouse EIAO DW release 2 is described. EIAO DW is a data warehouse that holds results from the European Internet Accessibility Observatory (EIAO) project. These results are mainly about the accessibility to disabled users of web resources that are automatically crawled and evaluated by other parts of the EIAO Observatory. However, the results also include statistics about technologies used by and linked to from the tested web resources.

A.1 Introduction

A.1.1 Brief Project Description

The overall objective of the project is to contribute to better eAccessibility for all citizens and to increase use of standards for on-line resources. The project will be carried out as part of the Web Accessibility Benchmarking cluster (WAB) together with the projects SupportEAM and BenToWeb.

The project will establish the technical basis for a possible European Internet Accessibility Observatory (EIAO) consisting of:

- A set of web accessibility metrics.
- An Internet robot for automatically and frequent collecting data on web accessibility and deviations from web standards (the WAI guidelines)
- A data warehouse providing on-line access to collected accessibility data.

A.1.2 Scope of this Appendix

This appendix covers the conceptual model for release 2.0 of the data warehouse in the EIAO project, the EIAO DW. The entire schema design follows the classic approach with conceptual, logical, and physical models.

A.1.3 Related Work and Readers' Instructions

This appendix is related to the following documents:

- EIAO Deliverable 6.1.1.1-2 [87] is the functional specification for the EIAO DW release 2.
- EIAO Deliverable 5.1.1.1-2 [111] is the functional specification for the EIAO crawler and describes (together with [75]) what data and how that data is collected.
- EIAO Deliverable 3.2.1 [75] describes the Web Accessibility Metrics (WAMs) that generate the data to store in the data warehouse.

A.2 Conceptual Model for EIAO DW

The conceptual model is shown in Figure A.1. The notation is based on UML 2.0 (see www.uml.org). The model illustrates classes for which information is to be stored in EIAO DW. In the model, attributes of the classes are shown as well as associations between different kinds of classes.

The dotted ellipses are strictly speaking not part of the conceptual model, but are included for ease. They show how the classes are grouped together as dimensions in the logical model. In ellipses, the hierarchy within each dimension is represented such that higher levels in the hierarchy are drawn above lower levels in the hierarchy. For example, in the Date dimension it is seen that dates roll up into months.

In the following, we describe each of the shown classes and its associations to other classes. For a detailed description of attributes, please refer to [88]. Whenever we refer to a class, its name will be capitalized. When we refer to an entity represented by an instance of a class, the entity name will not be capitalized.

TestResult Fact Table This is the class used to represent a single test of single subject with a single version of a specific barrier computation being used. For example, there will be an instance of TestResult for each time a specific img element on a given web page is tested with a version of a barrier computation that deals with img elements. The TestResult is associated with other classes. Specifically, TestResult is

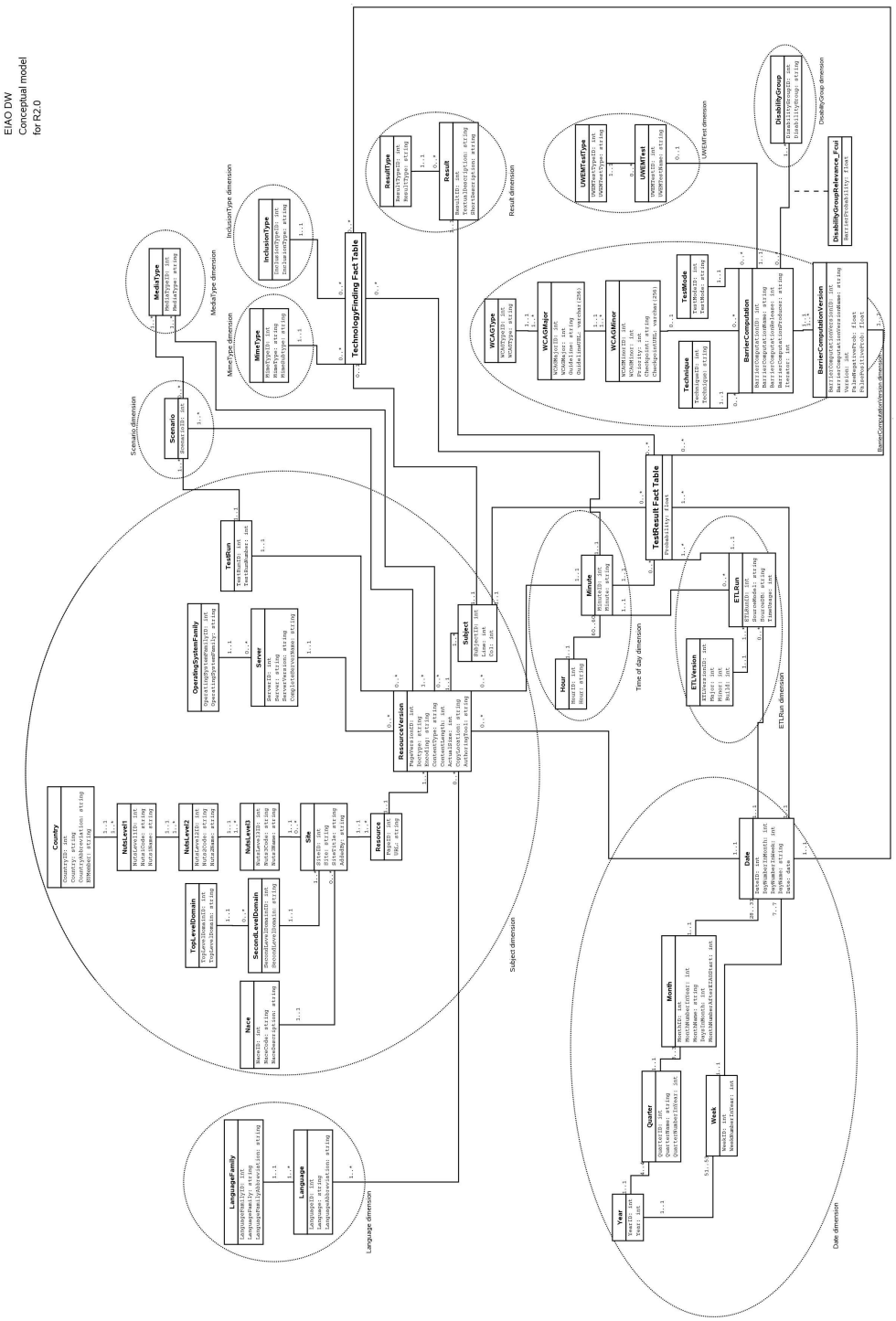


Figure A.1: The conceptual model

associated with the classes Subject, Result, and BarrierComputationVersion, describing what was tested, what the result was and what barrier computation version was used, respectively. TestResult is also associated with Minute and Date such that it can be represented when the test took place. Finally, TestResult is associated to ETLRun such that it can be represented in which ETL run a specific test result was created.

Subject The Subject class represents the tested subjects. The term subject is here used in the same sense as in [75]. Thus, a subject is a CSS or (X)HTML element that is relevant for a specific test.

Subject is associated to PageVersion. The association shows that a specific subject belongs to a specific version of a specific page. Other page versions may very well have identical subjects as well as a page version may (and probably will) have many subjects. However, one subject belongs to one and only one page version.

ResourceVersion The ResourceVersion class represents specific versions of web resources (such as HTML and CSS resources). For example, a page like the front page of http://news.bbc.co.uk/ is often updated and therefore different versions of this page will be considered in different surveys. Thus, ResourceVersion models the dynamic aspects of web resources (the static aspects are modeled by Resource described later).

A resource version is associated to a date and a timestamp. These represent when the resource version was last modified. A resource version has exactly one associated resource. A resource version also has exactly one associated test run. Thus, each time a resource is assessed, we consider a new version of the resource (which may or may not be identical to the previous version assessed). A resource version is associated to a number of media types that represent which media types the resource version contains explicit CSS rules for. A resource version is also associated with a number of scenarios. This is to represent in which scenarios the resource version was used. It is possible for, for example, a CSS file to be used in many different scenarios. Further, a resource version has exactly one server which represents the server hosting the resource. A resource version can have many associated subjects. A resource version is also associated with a number of languages to represent the natural languages used in the resource.

TestRun The TestRun class represents the surveys performed by the EIAO project. One test run can consider many web sites with many page versions. A test run can consider many resource versions and many scenarios. This is represented by associations with the ResourceVersion and Scenario classes.

Server This class represents the different kinds of server software (like Apache, IIS, etc.) used to host the web resources. One server product can host many page versions. A server product is associated with the operating system it runs on such that Apache for Windows is different from Apache for Linux. Thus, the Server class is associated to the OperatingSystemFamily class.

OperatingSystemFamily The OperatingSystemFamily class represents the different generic families of operating systems that the server products are running on. For example the Windows family covers both Windows XP and Windows 2000 as members. The Unix family covers all the different flavors of Unix, including Linux, FreeBSD, MacOS X etc.

Resource The Resource class represents web resources. However, the changing parts (such as the size of the content) of resources are represented by ResourceVersion. Resource only represents the static parts. Thus a resource can have several associated resource versions. A resource is considered as belonging to one site.

Site The Site class represents different web sites including their web addresses. A site is associated with a second level domain, a NUTS code (level 3) and a NACE category.

NUTSLevel3 The NUTSLevel3 class represents NUTS codes [38] at level 3, i.e., the lowest level. The class is associated with the class NUTSLevel2 for representing NUTS codes at the next level.

NUTSLevel2 The NUTSLevel2 class represents NUTS codes at level 2, i.e., the middle level. The class is associated with the class NUTSLevel1 for representing NUTS codes at the next level and with NUTSLevel3 for representing NUTS codes at the lower level.

NUTSLevel1 The NUTSLevel2 class represents NUTS codes at level 2, i.e., the highest level. The class is associated with the class NUTSLevel2 for representing NUTS codes at the middle level. Also, the NUTSLevel1 class is associated to the Country class.

Country The class Country is used to represent countries.

SecondLevelDomain The class SecondLevelDomain represents second level domains. A second-level domain can have several associated sites, but only one top level domain. Note that for "short" addresses such as relaxml.com, both the SecondLevelDomain and Site instances represent the entire domain name. For www.relaxml.com, the Site class represents everything of this address (including the www part) whereas the SecondLevelDomain class only represents the relaxml.com part.

TopLevelDomain Top level domains are represented by the class TopLevelDomain. A top level domain can have several second level domains.

Nace The class Nace is used to represent a NACE category that the owner of a site belongs to. The Nace class is associated with the Site class.

Language The Language class represents the different languages used on assessed web pages. A language may be spoken differently in different countries. For example, a language could be "German as spoken in Germany" and another "German as spoken in Austria". Therefore, the Language class is associated with the LanguageFamily class that represent the "generic" languages, i.e. "German" in the previous example. A language belongs to exactly one language family. Note that sometimes it is only possible to detect that a resource version uses "German" as language and not if this is "German as in Austria". For that reason, Language can also represent the generic language without any country information.

LanguageFamily The LanguageFamily class represents the language used when the "sub language" is ignored. For example for "en_US" which means "English as spoken in the US", the language family is "English". A language family has at least one member, but can have many.

Date The class Date is used for representing the day part of a specific date. The values will be added to the data warehouse on an on demand basis. The class Date is participating to the Date dimension in the logical model. Classes to represent dates (or a date dimension in the logical model) are used instead of using attributes of SQL type DATE. This is beneficial for many reasons. Since this class is the first of these classes, the advantages will be briefly described here. One reason is that it is possible to represent the value "Unknown". The integer IDs are also useful when precomputed aggregates are to be handled (e.g., the results for a specific year). Further, the use of classes (a dimension in the logical model) gives the possibility to represent domain specific knowledge that otherwise would have to be handled in the reporting layer. An example of the latter is the attribute MonthNumberAfterEIAOStart that for a given month gives the month number relatively to when the Observatory was launched.

Instead of using calender logic in the reporting layer, it is then immediately possible to see this. However, to provide a much flexibility as possible, an attribute of type DATE is also added to the Date class.

Date is associated with Month and Week. It is also associated with TestResult such that the date for an assessment can be tracked. Further it has an association to ResourceVersion representing that a resource version has been modified at a specific date. Thus the Date dimension is used as an outrigger from the Subject dimension in the logical dimensional model. In the same way Date is associated with the ETLRun class (to represent when a specific ETL run was started) and the Date dimension is, thus, also used as an outrigger from the ETLRun dimension in the dimensional model. Finally, Date is associated with TechnologyFinding. This is to represent when the use of a specific technology was found by the EIAO crawler.

Month The class Month represents months. A month belongs to exactly one quarter and has between 28 and 31 days. The values will be added to the data warehouse on an on demand basis.

Quarter The Quarter class represents calendar quarters. A quarter has 3 months, but belongs to exactly one year. The values will be added to the data warehouse on an on demand basis.

Year The class Year represents a calendar year. The values will be added to the data warehouse on an on demand basis. A year has four quarters and a number of weeks.

Week Calendar weeks are represented by the class Week. A week has a week number which is relative to exactly one year (even though a week may start in one year and end in another). The values will be added to the data warehouse on an on demand basis.

Minute The class Minute represents minutes. Minute is associated with Hour and ResourceVersion (the latter to represent the last modification time for a page version). Minute is also associated with the ETLRun class to represent when an ETL run was started. Thus, the Time dimension is also used as an outrigger from the Subject and ETLRun dimensions in the dimensional model. Further, the Minute class is associated with TestResult to be able to represent the time where an assessment took place and with TechnologyFinding to represent the time where a specific technology was found by the EIAO crawler. Note that all represented times are to be interpreted as UTC times.

Hour Hour represents hours. Note that all represented times are to be interpreted as UTC times.

ETLRun The ETLRun class represents specific runs of the ETL software. Each time the ETL is started for a given DW, an instance of ETLRun is created. This makes it easier to track the origin of data and to do statics about the amount of added data and time usage in each ETL run. The ETL program used has a version and exactly one ETL version is used by a specific ETL run. This means that ETLRun is associated with the ETLVersion class.

ETLVersion ETLVersion represents different versions of the ETL tool. This is useful if, for example, a bug is found in a specific version and all possible affected data should be located.

BarrierComputationVersion The BarrierComputationVersion class is used to represent a specific version of an implementation of a barrier computation (represented by the BarrierComputation class described below) as defined in [75]. BarrierComputationVersion is associated with BarrierComputation.

BarrierComputation The BarrierComputation class is used to represent a barrier computation as described in [75] (but disregarding the specific implementation version). The BarrierComputation class is associated to the UWEMTest class. This association represents which UWEM test the represented barrier computation is dealing with. BarrierComputation is also associated to the WCAGMinor class to represent which WCAG 1.0 checkpoint it is checking for. Further, it has associations to the TestMode and Technique classes to represent the mode of the barrier computation and what (CSS or HTML) it considers.

The BarrierComputation class is also associated to the DisabilityGroup class. This is to be able to represent how disability groups are influenced if a given barrier computation test does not pass. Therefore there is an association class for this association. This association class has the attribute BarrierProbability. That attribute is used to represent UWEM F_{pu} values (i.e., values that show how severe a failed test is for a disability group).

Technique The Technique class represents techniques covered by barrier computation versions.

TestMode The TestMode class represents test modes used by barrier computation versions.

WCAGMinor The WCAGMinor class represents a WCAG 1.0 checkpoint. A checkpoint is associated with (i.e. belongs to) a guideline (represented by the WCAG-Major class).

WCAGMajor The WCAGMajor class represents a WCAG 1.0 guideline. A guideline has a number of associated checkpoints (represented by the WCAGMinor class). WCAGMajor is associated with WCAGType to represent which type the WCAG checkpoint has (currently WCAG 1.0 or "None").

WCAGType WCAGVersion represents the version of the WCAG guidelines that a checkpoint belongs to.

DisabilityGroup The DisabilityGroup class represents disability groups (e.g., blind people, deaf people, and people with dyslexia) for the EIAO observatory. DisabilityGroup is associated with the BarrierComputation class. This association has an association class with the attribute BarrierProbability. This attribute holds the probability for that a subject introduces a barrier for the relevant disability group if the barrier computation fails when testing the subject. This will be used when computing aggregates by means of C-WAMs (see [75]).

UWEMTest The UWEMTest class represents UWEM tests. The class is associated with UWEMTestType that represents which types of documents the test handles. Further, UWEMTest is associated with the BarrierComputation class to represent which barrier computation that incorporates the represented UWEM test. UWEMTest only has one attribute apart from the ID.

UWEMTestType The UWEMTestType class represents types of UWEM tests represented by the UWEMTest class.

Result The Result class is used to represent the results of an EIAO assessment. A result belongs to a specific type of results (see below). There are many results describing the fails that resulted in negative outcomes of the application of the barrier computation versions, but only one pass result which is used for all positive outcomes.

ResultType The ResultType class is used to represent general types of results (tests can be passed or failed).

MediaType The class MediaType is used to represent the medias that can be supported explicitly by CSS. Thus the MediaType class is used to represent those media types and is associated with the Scenario class to represent which media types a specific page scenario found explicit support for. Similarly, MediaType is associated with ResourceVersion to represent which media types a specific resource versions supports explicitly. This is a many-to-many association as one media type may be supported in many page scenarios and one page scenario may support many media types.

Scenario The Scenario class is used to represent a scenario. It participates in a many-many relationship with the ResourceVersion class (as previously described this association represents which media types are supported within a single resource version). It is also associated with the TestRun class to represent which test run a specific scenario is used in. Finally, the Scenario class is associated with the MediaType class. This association is used to represent which media types are explicitly supported in a specific scenario (i.e. in the (X)HTML and the possible CSS files).

TechnologyFinding Fact Table This is the class used to represent the finding of a single subject that uses (i.e., holds or links to) an object using a specific technology. For example, there will be an instance of TechnologyFinding for each time a specific img element for a JPEG image is found on a given web page.

The TechnologyFinding class is associated with Subject (to represent the where the technology finding was done), MimeType (to represent the used technology), InclusionType (to represent how the technology is used.), Minute, and Date (the two latter to represent when a specific technology was found by the EIAO crawler).

Since TechnologyFinding has no measures, it is a so-called factless fact table used to track events.

MimeType The MimeType class represents the different MIME types found in resource versions.

InclusionType The InclusionType class is used to represent how a given technology/object is included in a resource.

Appendix B

Summary in Danish / Dansk resumé

Denne afhandling omhandler aspekter af specifikation og udvikling af data warehouse-teknologi til komplekse webdata. Store mængder data findes i dag i diverse webressourcer i forskellige formater. Men det er ofte svært at analysere og forespørge på de ofte store og komplekse data eller data om dataene (dvs. metadata). Det er derfor interessant at anvende data warehouse (DW) teknologi til disse data. Men at anvende DW-teknologi til at håndtere komplekse webdata er ikke trivielt og DW-forskningsmiljøet møder i den forbindelse nye, spændende udfordringer. Denne afhandling beskæftiger sig med nogen af disse udfordringer.

Arbejdet, der har ledt til denne afhandling, er primært sket i forbindelse med projektet European Internet Accessibility Observatory (EIAO), hvor et data warehouse til *tilgængelighedsdata* (enkelt sagt data om, hvor brugbare webressourcer er for handicappede brugere) er blevet specificeret og implementeret. Men også for andre projekter, der benytter business intelligence (BI) og/eller komplekse webdata, kan denne afhandlings resultater være relevante. En interessant vinkel på arbejdet dokumenteret i afhandlingen er, at både den benyttede og den udviklede teknologi er baseret på open source software.

I afhandlingen præsenteres flere væktøjer i en undersøgelse af mulighederne for at benytte open source software til BI-formål. Hver kategori af produkter evalueres i forhold til kriterier, som er relevante for brug af BI-produkter i industrien. Herefter beskrives erfaringer med at designe og implementere et DW til tilgængelighedsdata. Desuden præsenteres de konceptuelle, logiske og fysiske modeller for DW'et. Dette er så vidt vides første gang et generelt og skalerbart DW laves til tilgængelighedsområdet, som både er komplekst at modellere og at beregne aggregeringsresultater for.

Afhandlingen præsenterer også generelle interessante problemområder og løsninger dertil, som er fundet under arbejdet med at udvikle et DW og understøttende DW-teknologier til EIAO projektet. En ny og effektiv metode til at gemme tripler fra en OWL ontologi kendt fra Semantic Web-feltet beskrives. I modsætning til klassiske triplestores, hvor data gemmes i få, men store tabeller med få kolonner, spreder den præsenterede løsning data ud over mange tabeller, som kan have mange kolonner. Dette gør det effektivt at indsætte og udtrække data, i særlig grad i forbindelse med bulkload, hvor der er store mængder data.

En ny, nem og fleksibel metode til udveksling af relationelle data via XML-formatet (som f.eks. bruges af webservices) præsenteres også. Med denne metode spares arbejde med at programmere ofte komplekse løsninger til at håndtere udveksling af data korrekt. Med den præsenterede løsning skal brugeren kun angive, hvilke data der skal eksporteres og strukturen af den genererede XML. Dataene kan så automatisk eksporteres til XML og derefter importeres ind i en anden database, ligesom opdateringer af XML'en automatisk kan migreres tilbage til den oprindelige database.

Regressionstest er anerkendt og udbredt i forbindelse med softwareudvikling. I forbindelse med Extract–Transform–Load (ETL) software er regressionstest dog traditionelt en besværlig og tidskrævende proces. Afhandlingen udpeger specifikke forskelle mellem testning af "normal" software og ETL software, og på den baggrund introduceres et nyt semiautomatisk system til ETL test, der gør det nemt og hurtigt at iværksætte regressionstestning. Men den løsning kan regressionstest af ETL software begyndes på få minuter.

Traditionelt er DWs blevet bulkloaded med nye data på foruddefinerede tidspunkter f.eks. månedligt, ugentligt eller dagligt. Men en ny trend er at loade nye data, så snart de er til rådighed f.eks. i en web-log eller fra en anden webforbundet ressource. Dette gøres vha. SQL INSERT kommandoer, men disse er langsomme i sammenligning med bulkloadteknikker og databasesystemet begynder at yde dårligt. Afhandlingen præsenterer derfor en ny, innovativ metode, der kombinerer det bedste fra disse verdener. Data indsættes via INSERT kommandoer, men gøres først til rådighed i DW'et præcist når der er brug for det. Man opnår på den måde ydelse, som når bulkloading benyttes, men INSERT-agtig adgang til nye data.